FOUNDATION ELEMENTS FOR COMPUTER SOFTWARE SYSTEMS IN THE FLUID SCIENCES

by

ROBERT C. GAMMILL

B.S. The University of Rochester
(1959)

S.M. Massachusetts Institute of Technology
(1963)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1969

Signature of Author .....
Department of Meteorology, May 26, 1969

Certified by .....
Thesis Supervisor

Accepted by .....
Chairman, Departmental Committee on Graduate Students

FOUNDATION ELEMENTS FOR COMPUTER SOFTWARE SYSTEMS IN THE FLUID SCIENCES

by

ROBERT C. GAMMILL

Submitted to the Department of Meteorology on May 26, 1969 in partial fulfillment of the requirement for the degree of Doctor of Philosophy.

ABSTRACT

Software foundation elements are presented that are designed to allow effective utilization of FORTRAN application programs from a library. The interpreter, which manipulates character strings, allows transformation of input and output statements and controls the other elements. The hash coded associative memory stores and retrieves facts about library programs, calling sequences, data types or other pertinent information. The evaluator program carries out numerical evaluation of formulas, allocation of numeric storage, calling of library programs and other essential operations necessary for library utilization. The control of these elements is maintained by a set of rules to be interpreted. Those rules may be written by anyone, so systems constructed from these elements may be modified or extended by any user. Thus, the foundation elements form a basis for a class of extendable systems. Extendability of the systems is also available through additions to the library and the facts stored in the associative memory. Pertinence of this class of systems to the fluid sciences is centered in the ability to handle large multi-dimensional arrays of numbers, large and complex library programs, and to produce graphical output in the form of contour maps through the use of initially provided library programs.

Thesis Supervisor: Edward N. Lorenz

Title : Professor of Meteorology

TABLE OF CONTENTS

PREFACE

This thesis is a contribution to computer methods in the fluid sciences. The author, having substantial background in both computer science and meteorology, felt called upon to attack this fertile and relatively unexplored area. Computation is immensely important in the fluid sciences. Applications of computation, such as numerical modeling of the physical processes occurring in fluids or the statistical computations relevant to those processes, have received careful scrutiny by the most competent of researchers. The unexplored area is concerned with computer utilization and programming methods which will allow the many application programs produced by fluid scientists to be used by others. The development of such methods is crucial, because in the past most fluid science application programs have been written and used in ways that make them extremely difficult for anyone, other than the writer, to use. This has meant that, at least in computer programming, the work of the present has frequently been unable to build upon the work of the past.

This thesis presents basic programs and methods to be used in building and using application program libraries. These methods create an environment which encourages the authors of application programs to write them as general purpose utility subroutines. Those methods also allow the inexperienced user to operate application programs with little knowledge of their intricacies.

CHAPTER I

COMS: COMMUNICATION MANAGEMENT SYSTEM

1.1 Introduction

As the thesis title states, this thesis presents a collection of foundation elements (basic computer programs and methods) for computer software systems in the fluid sciences. These elements are the focal point of the presentation, but in order to demonstrate their usefulness it is necessary to actually implement a system or systems which use them to advantage. Those systems shall be called COMS, an acronym standing for Communication Management System.

The giving of a name to a class of systems may appear strange, but in fact names are often given to objects which change drastically with time. Examples of such objects are people, cities and companies. At various instants in their lifetime these objects are very specific and concrete, yet when different instantaneous examples of the same object are compared, little overt similarity may be found. In other words, John Jones as a baby bears little resemblance to the adult. Thus, when we attach a name to something which grows or changes, we pick some unifying and unchanging principle for attaching the name. For our examples those might be, the soul, geographic location and papers of incorporation. COMS is also an object which grows and changes, and the following definition tells how different instances of it may be recognized.

DEFINITION: COMS is any system, constructed from the foundation elements, whose primary goal is effective utilization of FORTRAN programs from a library.

COMS is the name for any system from a subset of the universe of systems which may be constructed from the foundation elements. The particular COMS's on which we will concentrate are those applicable to fluid science problems.

1.2 Description of COMS

COMS is constructed from three major programs. Those programs are:

1. An Interpreter - which serves as the control element for COMS.
2. An Associative Memory - which stores factual information about the library and COMS.
3. An Evaluator - which evaluates algebraic formulas, stores and retrieves numeric data, and causes execution of FORTRAN programs from a library.

Each of these three programs has an associated data collection. Those collections are:

PROGRAM	DATA COLLECTION
1) INTERPRETER	program (rules) to be interpreted
2) ASSOCIATIVE MEMORY	facts which can be retrieved

3) EVALUATOR COMS numeric variables and arrays.

Figure 1.1 shows a block diagram of these objects and the flow of information between them.

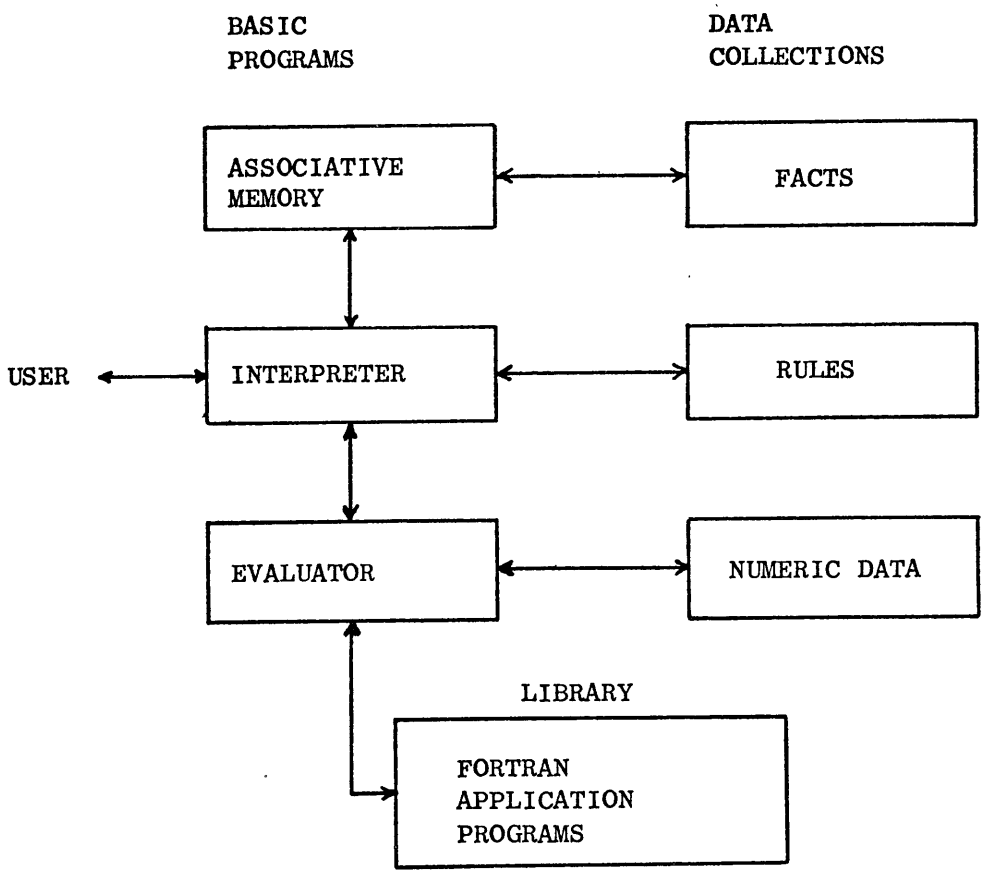


Fig. 1.1 Block diagram of basic programs and data collections, showing flows of information.

COMS is a system whose aim is to allow effective utilization of FORTRAN programs from a program library. Those programs are to be primarily user-contributed and oriented towards specific problem areas. Examples might be graph plotting, fourier transforms, or numerical models in fluid dynamics. COMS is constructed so that in most cases

few changes need be made in a program in order to add it to the library.

COMS is a data directed system, controlled by an interpreter, all of whose data can be created, modified or added to by anyone. Thus, COMS has been designed to be changed. Not only to be changed by systems programmers and by its designer, but by any and every one of its users. The only restriction upon modification is that original copies of basic COMS control information (rules or facts) cannot be changed except by system programmers. This is no restriction on anyone's ability to make changes, for he can make copies of any such data, and modify those copies to his heart's content.

Because of the freedom of change allowed for the data which drives the COMS system the programs which make up the interpreter should be much less subject to change. Because STRAN, the language of the interpreter, is very expressive yet simple in form, changes and additions to any version of COMS will be relatively easy to carry out.

COMS can be viewed as being analogous to a library (book storage type) and the methods and elements from which it is constructed can be related to the methods and elements of library science. As in library science, where a library cannot be effectively organized by piling a collection of unbound manuscripts in a room and allowing users to leaf through them, an effective program library cannot be organized in a haphazard manner unless it is to be very small and used by a small and knowledgeable group of users. As in conventional libraries,

the general user of COMS will be most concerned about what specific programs (books) from the library can do to achieve his goals. The ability or inability of the programs (books) to accomplish those goals will be seen by the user as defining the effectiveness of the library. However, the computer scientist (library scientist) must have a much different view, for if he takes that approach all of his time will be spent personally leading each user to the relevant programs (books). He may even end up trying to write programs aimed at the particular user's specific goals. That would be inefficient use of the scientist's time. Instead, he must try to create a system which allows the user to find and use what is needed on his own. He must also generate an environment which causes authors to write the programs (books) needed by users. This is the point of view which has been taken in COMS.

Thus, despite the fact that COMS is aimed at the fluid sciences, its library does not yet contain programs which will solve many realistic fluid science problems. That library does include a number of graphic display programs which will be particularly useful to fluid scientists. These might be regarded as analogous to basic reference materials (eg. a dictionary) in an ordinary library. This collection of graphics programs forms a basic library, which makes COMS relevant to the fluid sciences. Users may supplement the collection, in order to make COMS more relevant to their specific problems.

The library analogy has another valid aspect, which bears some further emphasis. Like a library, COMS has at least four distinct

(but not necessarily disjoint) groups of people that will be concerned with its operation in one way or another. These groups are the following, presented in analogous pairs:

	LIBRARY	COMS
1)	library scientist	computer scientist
2)	librarian	systems programmer
3)	author	application program designer
4)	reader	user

The scientist will normally be concerned with the study and design of functional elements to make a class of systems (libraries) work more effectively. The systems programmer (librarian) will be concerned with using the tools provided by the scientist to make a particular system (library) a functioning, effective and useable tool for users (readers). Finally, application programmers (authors) will be concerned with producing programs and data (books and manuscripts) to be placed in the system for general use.

The importance of recognizing the different groups described above stems from the fact that problem-oriented software systems of the past have rarely done so. The result has been that modifications to such systems have usually had to be carried out entirely by the original designers or their successors. This has been because the system was not designed with change in mind. Furthermore, application programs have had to conform to so many restrictive conventions and standards that at best existing application programs have had to be massively

rewritten to fit in the system and at worst they have had to be written from scratch. The result has been that such systems have remained alive only so long as the people who originally designed them continued to work on, add to, and develop the system. This has also meant that the development of such systems has been slowed to the pace that can be achieved by these people.

COMS has been designed with a much different point of view. That point of view states that the computer scientist should design the basic foundation elements of the system. These elements should be used by systems programmers to produce particular COMS implementations, including user language and system control information. Application programs should be submitted to the library by sophisticated members of the user population. Authors of application programs whould be rewarded, much as authors of books receive monetary rewards from their efforts. The result should be a system which will grow and develop much more rapidly.

1.3 Foundation Elements of COMS

In following sections a general description of the foundation elements which make up COMS will be presented. Each element will be described in terms of its purpose, its use, and its relationship to the other elements. This will be done so that in succeeding chapters, where more detailed examination of each element will occur, there will be some understanding of where the element fits in the total structure of COMS.

1.3.1 The Interpreter

The primary element of the COMS system is an interpreter that processes strings of characters. This processor serves as the control mechanism for the other elements of the COMS system. The primary function of the interpreter is to allow the definition of command languages for the user. The interpreter is directed by rules, presented as strings of characters, in its manipulation of data consisting of strings of characters. The rules may cause character strings to be passed to and received from the associative memory and the evaluator. The rules may also cause character strings to be collected from the input or placed in the output, communicating with the user. Since transformations may be carried out on the data by the rules, the interpreter serves as an intermediary and translator interposed between the user and the other elements of COMS. The interpreter and its language are examined in Chapter VI.

The interpreter is written in PL/1 to utilize the flexible character string manipulation capabilities of that language. The result is a compact and lucid interpreter which could be sped up by recoding in assembly language should that become necessary. The language of this interpreter, the "string transformation language" (STRAN), resembles COMIT (Yngve, 1962) in that it uses sequentially interpreted rules which decompose, transform, and recompose character strings. STRAN also resembles COMIT in that each rule passes control to one of two other rules, depending upon the success or failure of the decomposition portion of the rule. STRAN differs markedly from COMIT and

other character manipulation languages in that no distinction is made between rules (procedures) and data. A character string to be interpreted as a rule must simply conform to certain rules of format if successful interpretation is to occur.

1.3.2 The Evaluator

The evaluator serves as a mechanism for evaluating algebraic formulas. Also important is its duty as an interface mechanism to the FORTRAN programs in the library. The evaluator is directed by character strings passed to it from the interpreter. The numbers with which it deals may be treated as floating or fixed point constants, variables or arrays. In the evaluation of formulas, most of the normal built-in functions of FORTRAN (eg. SIN, TAN, ABS), and all of the operators (eg. +, -, **, etc.), may be used. Two basic FORTRAN statements can also be evaluated. The first, the declaration statement, allows any variable or array to be declared INTEGER or REAL, causing dynamic allocation of space by PL/1. The second is the CALL statement, allowing loading and execution of programs from the user library.

All of the arguments of CALL statements and the dimensions of declarations may be equations to be evaluated. All variables, arrays or elements of arrays may be referred to by value or by name (address). The latter ability is primarily of use in argument lists for library routines.

A special capability of the evaluator allows character strings from the interpreter to be saved in a "namelist interface program" until sought by a namelist read in a library program. This capability allows especially flexible control of library programs from COMS. The use of this method of communication is described in Section 3.5 and in Appendix A.

The results of evaluations are always returned to the interpreter as character strings, so that the interpreter never deals with any data which is not a character string. The evaluator is examined in some detail in Chapter III.

1.3.3 The Associative Memory

The hash-coded associative memory for n-tuples of character strings allows factual information about COMS, the user, the library and other parts of the environment to be quickly and easily stored and retrieved. Use of the associative memory permits definition of phrases (user commands) to be translated into internal command forms and retrieval of information for users and for COMS itself. In very simple COMS implementatibns, where all such information may be assumed known by the user, it is possible to dispense with the use of the associative memory entirely. Many of the problem oriented systems to be described in Chapter II do not, in fact, have an equivalent mechanism. Those systems are restricted due to that lack. Description of the use of the associative memory is given in Chapter VII. Its implementation is discussed in Appendix C.

1.3.4 COMS Information Collections

In Section 1.2 we mentioned three distinct data collections, each associated with one of the three basic programs of COMS. Those collections were:

(a) character string rules and data for the interpreter

(b) n-tuple data for the associative memory program (optional)

(c) numeric variable and array data for the evaluator (optional).

Two of these data collections are optional, which means that for a particular COMS execution, core storage may or may not be allocated for that collection at the users option.

Each of these data collections may receive information from or pass information to secondary storage (eg. disk or tape). Both (a) and (b) may be read from or stored on secondary storage devices by a single command to the interpreter. That command specifies both the dataset and member name of a member of a partitioned dataset (terminology courtesy of Operating System 360). Numeric variable and array data to be read or written must be handled by library programs. This is necessary because formats of user data cannot be standardized as easily as those of rule and n-tuple data. The user may deal with as many of his own data collections in secondary storage as his needs dictate.

The user may add rules to a rule set in core by giving them as input when the interpreter is in its rule and command reading mode. Any other communication between the user and a data collection must

be controlled by the interpretation of rules, or by a library program.

The remaining COMS information collection is the program library. This is a partitioned dataset, which resides in secondary storage. The members of this collection are placed there, and given names, using the facilities of the linkage editor (loader). These members may be executed by passing a CALL statement with the appropriate member name to the evaluator.

For a detailed overview of the various elements of COMS which have been described, Fig. 1.2 will serve as the most suitable summary.

1.4 Basic Fluid Science Graphics Elements in the COMS Library

The elements of COMS, described in preceding sections, do not particularly limit or even aim the system toward the fluid sciences. They do fulfill the basic requirements of any system for the fluid sciences, that it be able to handle large collections of multi-dimensional numeric data and to execute large and complex programs coded in FORTRAN. But these requirements occur in many areas besides the fluid sciences. The place where COMS is seen to be oriented towards the fluid sciences is in the set of graphics programs provided as "seed" elements for the library. Besides the basic graphing programs provided in this library, there is a powerful collection of author written programs which are especially suited to solve the graphics display problems of fluid scientists. These programs produce contour maps on any available type of output device. Another program in the collection produces world

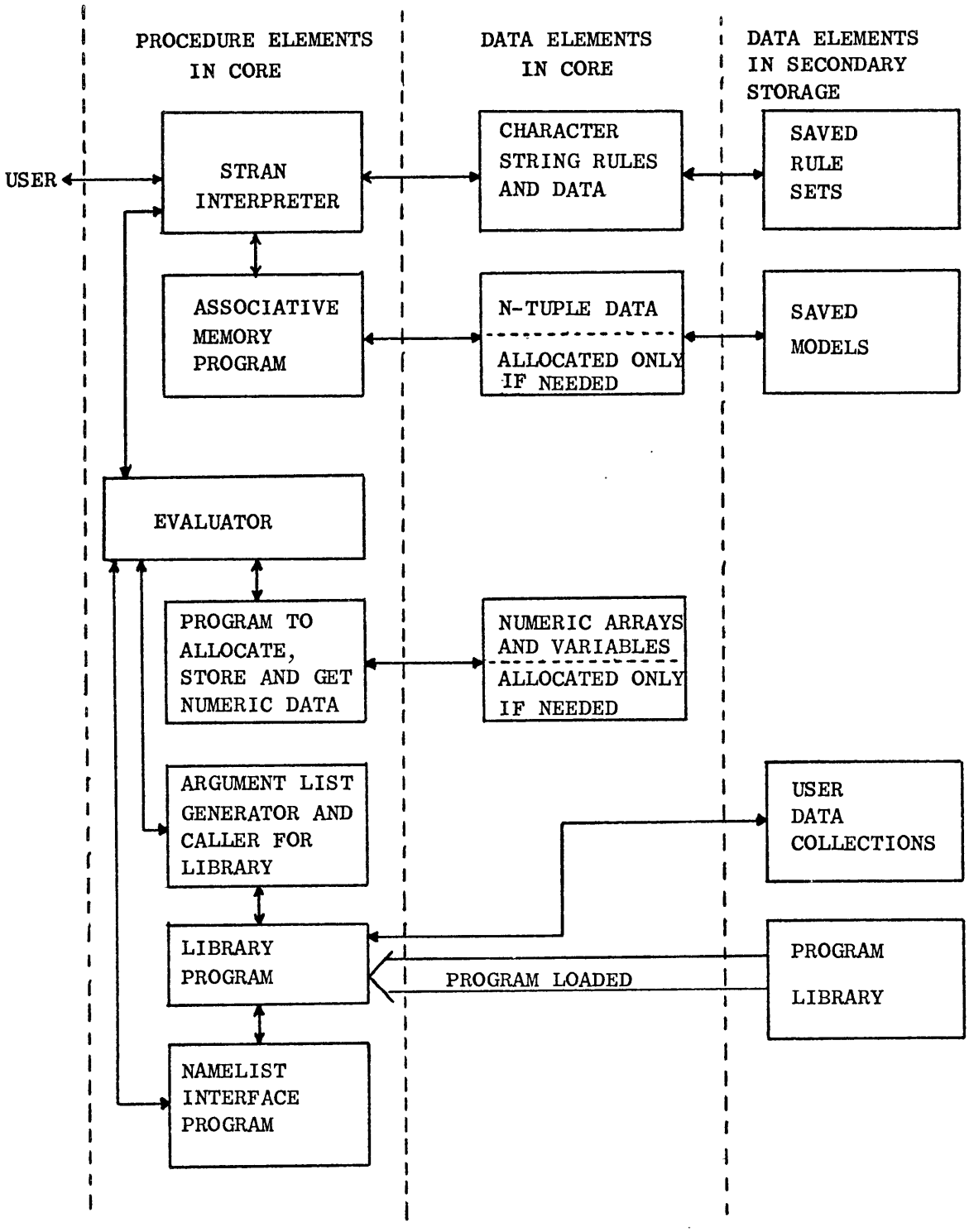


Fig. 1.2 Block diagram of COMS elements
(Arrows indicate exchange of information.)

geographic outline maps in a number of frequently used projections, to allow the fluid scientist to plot contour maps over actual continental outlines for physical reference. The methods used by these programs are described in Appendices D and E. These "fluid science" graphic tools, plus the ability to build and use his own library of application programs, provide the fluid scientist with a system that is well adapted to his needs.

CHAPTER II

A REVIEW OF SOME PROBLEM ORIENTED SOFTWARE SYSTEMS

In this chapter capsule summaries of other software systems will be presented. The systems presented range over a broad spectrum of problem areas, goals and implementation environments. Each is presented because at least one aspect of its approach contributed in some way to the development of the ideas inherent in the work to be described here. Some criticism of each system is presented, but this should not be taken to mean that they are somehow defective. Rather, each of those systems is a significant contribution to its particular problem area. The criticisms are presented to point up the reasons why each of them is lacking or limited with respect to the fluid sciences problem area. Only one of them, ANAL 68, was designed with that area in mind.

2.1 SKETCHPAD

SKETCHPAD (Sutherland, 1963) is an interactive graphical design system programmed in assembly language on the TX-2 computer at Lincoln Labs. Its striking feature is its creation of extremely flexible and complete models of the graphical objects and constraints being manipulated and use of highly intuitive primitive operations for manipulation of the objects modeled in nearly any consistent manner desired. Its failings, from the point of view of the meteorologist, are its

dedication to the particular problem space, the esoteric character of the computer and language in which it was implemented, and the near impossibility of user created extensions to its capabilities.

Another limitation of SKETCHPAD is that despite having a sophisticated modeling capability, it uses those models only for its graphic constructs and their constraints. SKETCHPAD does not use its models to help the user to learn to use it. Ease of learning to use this software system is based solely on its presumed external simplicity and logical consistency of approach.

2.2 MAP

MAP (Kaplow, 1966) is an interactive language for mathematical analysis within CTSS (M.I.T. IBM 7094 time sharing system). It has a wide range of highly flexible operations which can manipulate any desired named data object. Graphical display operators are available, and are quite flexible. The user can create and add his own operations either as sequences of commands in the MAP language or as compiled code in MAD. The MAP system is primarily designed for interactive use. However, it can be used in a noninteractive environment. Failings of the MAP system are its inability to handle two dimensional (or higher) arrays of numbers, and the associated graphics display problems, the fact that it was not written in a common language (eg. FORTRAN IV), and most important that it does not make use of any sophisticated modeling or fact storage and

retrieval techniques to aid itself and the user. In this last sense, the MAP system seems a step backward from SKETCHPAD, although such a judgement is difficult about two systems aimed at different problem spaces.

The MAP system gains a great deal of its flexibility through its use of the CTSS user file directory for cataloguing of named data objects (arrays and variables) and programs. The user file directory becomes a symbol table for the MAP system. This is an abnormal use of the file directory, at least in the frequency with which the directory is referenced, and causes much of the inefficiency which exists in the MAP system. This inefficiency is mitigated somewhat by the fact that this use of the file directory makes all of the users application programs accessible to the MAP system. Thus, the user can extend the application facilities of MAP in any way he desires. Another facet of MAP is that the application programs which are built-in (reside in the system's central file directory) contain interactive programming to instruct the user in the use of the particular program, and the arguments he may give it. This is a good feature, but requires considerable programming for each application program. Furthermore, it is not really a system feature, but one added to each individual application program. Such a feature, in this form, can be added to any system which allows execution of user application programs.

2.3 ANAL 68

ANAL 68 (Welsh, 1967) is an interpretive processor written in FORTRAN IV to operate under batch processing on the Univac 1108. It is tailored for a particular class of meteorological problems, and as result it operates on data objects which are a small number of fixed size two dimensional arrays of floating point numbers stored in common. New user created operators could be written in FORTRAN IV by a user who knew the numerous conventions and restrictions which must be followed. However, the new operator cannot be added dynamically for the whole system must be reloaded with the new operator included. Contouring programs are available, as are most of the usual numerical operators used by meteorologists. Thus, despite being much less flexible than either of the preceding systems, ANAL 68 is much more usable for the average meteorologist than either of them. Like MAP it has little capability for modeling or sophisticated information storage and exchange within the system. It appears more difficult to learn to use ANAL 68 than either of the preceding systems.

ANAL 68 transmits information to its application programs via a large named common block. All such programs and data areas must be loaded and in core throughout the systems operation. This severely restricts the amount of data and programs which may be handled by the system, but does achieve fast execution.

2.4 ICES

ICES (Roos, 1967) is a system developed by the Civil Engineering Department at M.I.T. It is a large, complex and costly system to which many man-years of effort have been applied. Like ANAL 68 and MAP, ICES interprets user commands, modifies the values of internal variables, and executes application programs. ICES is somewhat more general, however, in that it executes a command by interpreting code (resembling assembly language instructions) associated with that command name by a definition. Commands may cause the execution of application programs which are part of the subsystem of ICES being utilized. Subsystems, associated commands, and the application programs included can all be user created.

The major disadvantage of ICES is the fact that application programs to built into a subsystem must be coded in FORTRAN E (a subset of the full FORTRAN IV language), must include numerous special statements peculiar to ICES in order to maintain compatibility with the system programs, and must use a common block to receive information from the subsystem and the command which caused the program's execution. Not only does this mean that normal application programs must be massively rewritten to fit in a subsystem of ICES, it also means that different subsystems are incompatible since each will have a different common block organization. Like MAP and ANAL 68, ICES does not utilize any central system modeling capabilities, storing facts (data) about system operation and environment for

aiding users and maintaining effective operation. This statement should be softened by noting that the command definitions for each subsystem in ICES do form a sort of rudimentary model for that subsystem.

2.5 SIR and STUDENT

These two software systems were developed as problem solving systems for Ph.D theses, by Raphael (1964) and Bobrow (1964) of the artificial intelligence group at M.I.T.'s Project MAC. Both were concerned with solving problems (simple logical and semantic problems for SIR and high school algebra word problems for STUDENT) which were posed in simple English sentences. While they are philosophical predecessors of the work to be described here, these two systems were not dedicated to a problem environment which bears much external resemblance to the fluid sciences. Their contribution is to be found in the good use they were able to make of lucid models of their own particular problem spaces. This, and the power provided by the modeling capabilities in SKETCHPAD (Section 2.1), provided the impetus for provision of modeling capabilities in foundation elements to be presented here.

SIR and STUDENT were programmed in LISP, much too abstruse a language for an ordinary meteorologist user, and neither allowed the user to enhance the system by adding his own programs to a library. Both of these systems demonstrated a good ability to model real-world situations and facts, although SIR lacked the ability to model

relations between its relations. As will be seen in the material to follow, these two systems provided a concept of how a general purpose problem solving system should appear in its communication with a user, and the kinds of facilities necessary to accomplish that goal.

2.6 Comparison of COMS to its Predecessors:

COMS, like the systems described in preceding sections, is aimed at the user who knows little of computers, but is knowledgeable in the problem area of interest. This user can make good use of computer power if he can manage to communicate with the computer concerning the problem area.

Of the systems described only ANAL 68 is tailored to fluid sciences problems. It is a relatively inflexible system interpreting one fixed format operator per card and operating only in a batch processing environment. ANAL 68 requires considerable learning of conventions and restrictions by the prospective user since it vaguely resembles a machine language for manipulating large two dimensional arrays. Listings of command sequences in this language are readable only by an experienced user.

MAP, ICES, and SKETCHPAD are not tailored for meteorology (in fact SKETCHPAD is unusable for fluid sciences problems) but they are problem oriented systems which are flexible and fairly intuitive seeming to the user. An uninformed user can learn to use these systems

quickly by experience and occasional reference to a manual for appropriate names and commands.

COMS can carry out the same sort of fluid science oriented operations as ANAL 68 but user written additions to the program library are much easier to write. COMS is superior to MAP, ICES and ANAL 68 in its ability to maintain a model of the problem environment, which allows it to aid the user in controlling his application programs. The model will also allow COMS to carry out some of the self teaching operations of MAP, without requiring each application program to carry out lengthy self description, as is the case in MAP. Another advantage of COMS, which it shares with MAP and ICES, is its ability to operate in both batch and man-machine interaction environments. The motivation for such an ability is the economy which must be achieved for many time consuming jobs in meteorology.

Another advantage of COMS is that it is completely compatible with FORTRAN IV user programs, since almost all meteorological application programs are written in that language. It should be particularly emphasized that COMS can execute almost any subroutine coded in FORTRAN IV, without its being modified. That is a capability that exists in none of the preceding systems, although MAP comes the closest, being able to accomplish that with many subroutines coded in MAD.

COMS appears to be more flexible than its predecessors and attempts to incorporate the good features that can be found in each. The primary liability of COMS, at its present stage of development, is its relative inefficiency.

CHAPTER III

THE EVALUATOR - INTERFACE TO OS360

3.1 Background

The evaluator serves as an interface mechanism for the interpreter, to the world of numeric data and FORTRAN programs. The jobs of the evaluator are the following:

(1) Evaluating algebraic formulas.

(2) Allocating space for numeric variables and arrays.

(3) Storing values in and retrieving values from those variables and arrays.

(4) Generating argument lists in standard OS360 form, and passing the list to a program, that is brought to core from the library if it is not already in core.

(5) Providing a mechanism for passing character strings to the FORTRAN namelist routines where they may later be collected by a namelist read issued by a library program.

The evaluator receives directions from the interpreter as character strings, and returns any results as character strings.

The division of labor between the three major elements of COMS makes it quite easy to implement a very simple system which, in essence, uses only the evaluator. Such an implementation does not use the

associative memory and uses the interpreter only to read input lines, pass them to the evaluator, and print results returned. An implementation of this kind can be very useful to a knowledgeable user, and the capabilities of exactly such a simple version of COMS are demonstrated in Chapter V.

It is not possible to produce an implementation of COMS which does not use the evaluator. It is the most essential element of COMS for achieving the goal of effective utilization of library programs. The evaluator is carefully programmed, in PL/1 and assembly language, to be as general purpose as practical. However, it is much more computer dependent, and as a result less easily modified, than the other elements of COMS. This is so because it deals with the practical and concrete difficulties of working with Operating System 360. Because the evaluator deals with the harsh realities of library programs and collections of numbers, it is more dedicated to the specific problem area than either of the other basic programs of COMS.

In the remaining sections of this chapter we will examine the features of various parts of the evaluator. In the following chapter we will show exactly how the evaluator works with the program library.

3.2 The Evaluation of Formulas

The evaluation of formulas is carried out in two passes. The first pass accomplishes the collection of contiguous characters into symbols, and interpretation of those symbols as numeric constants, variable names,

array names, built-in function names, or operators. Also in the first pass the values of simple variables are retrieved and a precedence is assigned to each operator. Formulas are accepted in infix notation, with parenthesization optional in most cases, just as in FORTRAN. The second pass carries out a translation to a prefix Polish notation under control of operator precedence and parentheses. Evaluation of operators, calling of built-in functions and retrieval of values from arrays all occur during the second pass. Translation to prefix Polish means that lists of indices for arrays and arguments for subroutines will appear in sequence when it is time for them to be utilized.

Formulas for evaluation may include an equal sign, to cause assignment of the resulting value to a variable or array element. If an equal sign does not appear, the result of the evaluation is simply returned to the interpreter in the form of a character string. If an equal sign was included, the character string returned to the interpreter includes the name of the target variable and the equal sign as well as the value assigned.

The names of the built-in functions and operators which are available in the evaluator are:

+,−,*,/,**,unary+,unary−

MOD,FLOAT,FIX,ABS,SIN,COS,TAN,ATAN,EXP,LOG,LOG10,SQRT

When the mode (fixed or floating point) of the numbers going into a calculation are mixed, the fixed number will be floated. If a fixed point number appears as the argument of a built in function requiring a floating point argument, that number will be floated.

3.3 Management of Variables and Arrays

Storage for variables is dynamically allocated by assigning a value to the variable (eg. VARBL=5.0). The mode (fixed or floating) of the variable is defined by the mode of the number assigned. Storage for arrays is dynamically allocated by passing a statement of the following form to the evaluator:

(a) INTEGER NAME(first dimension, second dimension,...)

(b) REAL NAME(first dimension, second dimension,...)

NAME is the name of the array being declared. As many dimensions as desired may be given. Each dimension may be stated as a formula, including statement as a variable, or constant. If no variables or arrays are mentioned, no storage will be allocated for that purpose. Allocation of storage is handled by PL/1.

Values are stored in variables or array elements by assignment statement. Retrieval of a value is caused by the appearance of the variable name, or array name plus indices, in a formula. All indices for arrays and arguments for subroutines may be formulas to be evaluated.

3.4 Execution of library programs

Library programs can be executed by passing a CALL statement (described in detail in Chapter IV) to the evaluator. First, formulas occurring in the argument list (if any) are evaluated. Then, the name of the program to be called, and the list of results of formula evalu-

ations, are passed to an assembly language routine. That routine generates a standard OS360 list of argument addresses, if any arguments are to be passed. Then the name of the routine to be called is compared against a list of program names which are not named members of the program library, but which are present in special modules and may be executed through the CALL facility. These include Calcomp library service routines contained as entries in a special load module, a FORTRAN library service routine for error processing contained in another special load module, a PL/1 snap dump which is always loaded with COMS, and a CLOCK program which returns the time of day in hundredths of a second as an integer. If the name of the program to be called is not in the list of special routines a LOAD service macro (OS360 supervisor call) is issued. This causes the requested library program to be loaded into core, if it is not already there, and returns the entry address which is used to pass control to that program.

3.5 Namelist Interface and I-O

A special capability has been provided in the evaluator. This special capability allows character strings to be passed from the evaluator to a special namelist-read interface routine, to be saved till collected by the next application program issuing a namelist read. Any character string, starting with an ampersand, passed to the evaluator will be handled in this manner. When the collection of character strings saved by the interface routine has been exhausted (or immediately if no such strings have been saved) the namelist read will collect

input lines from the FORTRAN input device specified in the read statement. Thus, when no strings have been saved, namelist read works exactly as it does outside of COMS.

Normal operation of namelist read requires an ampersand to appear in column two of the first line of input, followed immediately by the namelist name. Normal operation is also such that column one of every input line will always be ignored, and the reading of input will be terminated by the characters &END. The following is an example of Fortran statements to carry out namelist read, and an example set of input lines.

```
FORTRAN STATEMENTS:  (DATA is the namelist name)
     NAMELIST/DATA/ARG1,ARG2,ARG3,ARG4
     READ(5,DATA)
INPUT LINES:  (lines begin at column 2)
     &DATA ARG1 = 5.0, ARG3 = 57.865,
     ARG2 = 77.8E-5   &END
```

These statements will cause the specified values to be stored in the FORTRAN variables ARG1, ARG2 and ARG3. In COMS the necessity for the inclusion of the names following the ampersands has been removed. Thus, if the same lines were to be passed to the evaluator by the interpreter, to be collected by the same program (now being executed from the COMS library), they would appear as follows.

INPUT LINES: (lines begin at column 1).

```
&& ARG1 = 5.0, ARG3 = 57.865,
 & ARG2 = 77.8E-5 &
```

The ampersands in column one cause the evaluator to transmit the character strings to the namelist interface program, and the remaining ampersands serve to initiate and terminate the sequence of variable names and values for the actual namelist input operation.

The utility of this method for transmission of values between COMS and a called program and the organization of the called program for maximum effectiveness is examined in Appendix A.

CHAPTER IV

THE PROGRAM LIBRARY, ITS ORGANIZATION AND USE

4.1 Restrictions on Library Programs

We have stated in earlier chapters that any FORTRAN IV subroutine can be placed in the library and utilized. However, there are some restrictions that must be understood. These would probably not be termed restrictions by a knowledgeable FORTRAN programmer and user of OS360 (Operating System 360), but they are reasons why certain types of FORTRAN subroutines will not be useable without modification. The primary restriction is:

(a) Any method of passing arguments to a subroutine which depends upon the subroutine having been linkage edited with the calling program, into an executable load module*, cannot be used.

This means that since the subroutines in the library are not linkage edited with COMS, they cannot receive arguments through a common block shared with COMS. This sort of requirement is normal for a utility subprogram, where "independence" from the calling program must be maintained. However, it is not true of just any randomly chosen subroutine. Another restriction is:

---

*A load module is any collection of programs whose interprogram references have been resolved, and can be executed without any further processing.

(b) Each library routine may have up to six entry points, but in actual practice it is better to have only one entry point per library program.

This restriction arises from OS360, which allows up to six names associated with entry points of a load module in a library, but when the module is moved into core, only remembers the single name by which it was requested. Thus, a later request for another of the program's entry names causes a second copy to be brought into core, instead of utilization of the entry already in core.

(c) If a library program is to carry out FORTRAN input or output, or call service subroutines which may do so, a special dummy subroutine must be loaded with the program when it is put in the library.

If each individual load module needing to carry out input-output activities were to carry them out itself, a great deal of confusion and duplication of effort would occur. For this reason a special load module containing all of the FORTRAN I-O and error processing routines is included in the library. Access to this load module by a library program is provided by including a special dummy program, containing all of the relevant service routine entry points, when the library program is linkage edited. A similar dummy program is necessary to gain access to a load module containing all of the Calcomp plotter library routines. When called, the dummy program issues a load macro, and collects the correct entry addresses from

the target load module. The inclusion of these dummy routines is the final requirement upon library programs in COMS.

The above described restrictions and requirements mean that some care must be exercised in putting a user program in the library. It also means that most general purpose utility programs can be put in the library without any modification whatsoever.

4.2 Library Load Module Organization of COMS

As was mentioned in the preceding section, the COMS library contains not only load modules created by users, but two special load modules, one for handling FORTRAN I-O and the other for producing Calcomp plotter output. Furthermore, the COMS interpreter itself is contained in the library and uses the FORTRAN I-O load module to do its own input and output. Thus, when a user program which produces Calcomp and FORTRAN output is in use, the set of load modules present in core will be as shown in the diagram below.

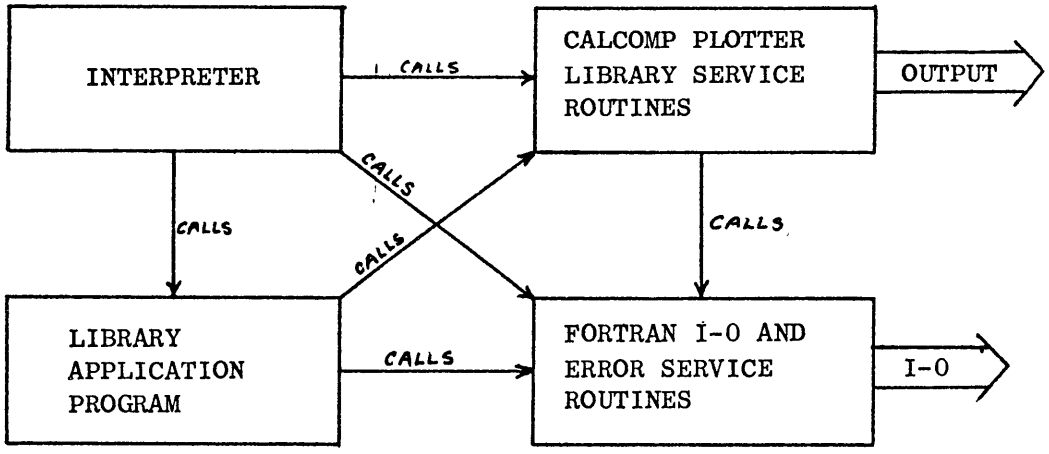


Fig. 4.1 COMS load modules

Most of the limitations of COMS upon programs in the library are caused by the fact that OS360 does not have efficient dynamic linkage resolution capabilities, as does a system such as MULTICS. Under OS360 the linkage editor would have to be called up each time a new user program was to be brought into core, if all linkages were to be resolved in the normal manner. The dummy program method of linkage resolution is a simple and effective, although perhaps not esthetic, solution to the problem.

4.3 Methods of Interprogram Communication

In Section 4.1 it was mentioned that communication of information between the interpreter and library programs could not be carried out via a FORTRAN common block (a method by which two or more programs may share data storage space). The question then arises as to what methods may be used. Two methods are available. The most straightforward of the two is the old standard, a list of arguments passed along with the subroutine call. This argument list is generated in standard OS360 form by COMS, so any program using those conventions can be called from COMS. Furthermore, by using certain coding methods within the called library program, it is possible to allow varying length argument lists to be passed. That method is examined in Appendix A. The form of the statement which will cause a library program to be called, and the methods of specifying arguments to be passed will be examined in the following section.

The second method of communication between the interpreter and a library program is through the NAMELIST interface mechanism described in Section 3.5 and Appendix A. Normally namelist input operates from an external input device to a program containing a namelist read statement. However, in COMS, the interpreter is able to transmit character strings through the evaluator to the library program to satisfy the namelist read. This allows assignment of values to specified library program variables by the interpreter. Furthermore, no modification whatsoever is necessary for the namelist read in the library program. When it is not being used as a COMS library program, the namelist read will operate in the usual fashion.

4.4 Calling a Library Program

A library program, whose member name (an OS360 term describing the name it was given when being put in the library by the linkage editor) is PROGNAME can be called by passing the following character string to the evaluator.

```
CALL PROGNAME
```

In this form no arguments will be passed to the program. The statement can also be written:

```
CALL PROGNAME(argument list)
```

The argument list may contain up to thirty arguments, separated by commas. The arguments themselves may be nearly any type of argument

that would normally appear in a FORTRAN program. They may include equations to be evaluated, character string literal constants, array names or specific array elements, as well as simple numeric variables and constants. Array elements and variables will normally have only their values passed, that is, the address passed will be that of a temporary location containing the present value of the variable or array element. Thus, a special method is provided for specifying that such arguments are to be passed by name (the actual address of the variable or array element to be used). The following list of examples should clarify these statements. In these examples we assume that the declaration REAL PSI(47,51) has previously been passed to the evaluator, stating PSI to be an array of floating point numbers.

(1) 'M6350' is a character string literal constant. Single quotes may be included in the quoted string by preceding each by another single quote.

(2) PSI is an array name, to be passed by name (the actual address of the array is passed).

(3) PSI(21,17) is an array element, to be passed by value (the address of a temporary location containing the present value of PSI(21,17) will be passed).

(4) PSI(I,J) is an array element (the present values of variables I and J are used) to be passed by value.

(5) @PSI(I,J) the same as (4), but the actual address of the particular element will be passed (by name) due to the @.

(6) I is a variable to be passed by value (address of a temporary location containing its value).

(7) @I is a variable to be passed by name.

(8) SIN(I)*J is an equation to be evaluated and the resulting value passed. An @ would cause an error since no variable or array is available to be specified by name.

(9) 7.9E-5 is a constant to be passed by value.

CHAPTER V

EXAMPLES USING A SIMPLE COMS

5.1 Basic COMS Implementation

In this chapter we will present a number of examples which demonstrate the capabilities of the evaluator and the program library. To do this we use an extremely simple rule set for the interpreter which collects input lines one at a time, separates each line into statements by finding semicolons, and successively evaluates those statements. The results of the evaluations are printed as output.

This rule set does not exhibit the capabilities of either the interpreter or the associative memory. Instead, it is an extremely direct and efficient way for a knowledgeable user to utilize the evaluator and the program library. The examples presented show how carefully written library programs may be used in a very flexible manner to carry out a variety of tasks. Since the rule set carries out no translations on the input statements, these examples show exactly what kinds of statements are accepted by the evaluator.

In Chapter VIII we will present more sophisticated rule sets for the interpreter, using the associative memory and translating user input statements to acceptable directions for the evaluator. Examples will be presented there to show how modeling and translation can make the facilities of the evaluator and library available to less knowledgeable users. Those examples will demonstrate how much

more powerful the interpreter and associative memory make COMS, as compared with the examples to be presented here.

5.2 Example 1

This example shows the capabilities of the evaluator, and two author written graphic display programs from the library. The first program, CALTUR, produces contours of a data field read by a simple input program, READ. The second program, GLOBE, produces the outlines of northern hemisphere continents for physical reference. The result of the example is the Calcomp plot shown on a following page.

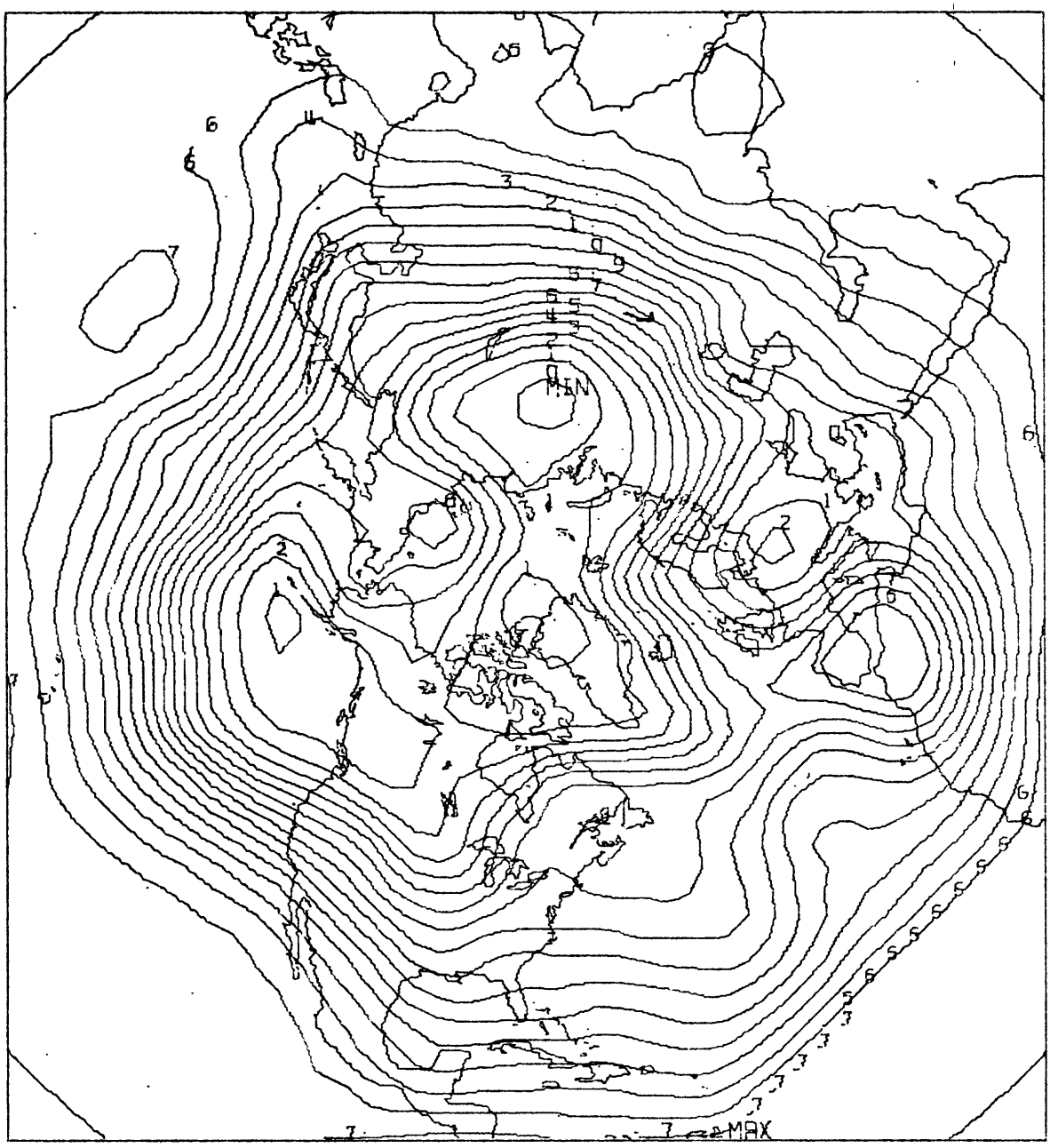
The first line of input for this example is a character string for the namelist interface mechanism, to be read by the contouring program. It sets the sizes of labels for the plot. The following input line gives values for COMS variables, and allocates space for the array PSI. Next, the READ program is called to read cards into PSI from the input stream. The cards were located in the input, immediately after the call to READ, but do not show in this listing because they were read by that program, using the format given in the calling sequence. The call to NEWPLT initiates the Calcomp plotter tape, and puts the users problem number and programmer number on it. Calls to CALTUR and GLOBE follow. The GLOBE program plots a projection of a map which is a user data set, stored as sequences of latitude-longitude pairs. Finally, the call to PLOT1 moves the Calcomp pen fourteen inches down the paper, and the call

to ENDPLT terminates the plotter tape.

The READ program, written in Fortran and located in the library, is presented as the final page of this example. That program uses a variable length calling sequence, examined in Appendix A. READ accepts the tape number, input format, array location, and beginning and ending indices of values to be read for the array. If the indices are not given only one value will be read. The READ program treats all arrays as if they were one dimensional, but this does not cause difficulty for multidimensional arrays so long as the information in the array is contiguous (all dimensions are fully utilized). Such a requirement is not difficult to satisfy when dimensions are being specified at execution time.

In the computer output which follows, lines which begin INPUT... have had that marker added to indicate that they are an echo of an input line. All other lines, with the exception of the heading which appears at the top of each page, and the statement "BEGIN INTERPRETING RULES." which indicates the start of rule interpretation, have been generated by rules, or by a library program. Outputs from library programs will always project out from the left margin, because all output handled by the STRAN interpreter is deeply indented.

```
STRAN INTERPRETER ON MAY 23,1969 AT 03:54:44.300 PAGE  3


BEGIN INTERPRETING RULES.


INPUT...&& FLAB=0.06,SIZE=0.08 &;
          && FLAB=0.06,SIZE=0.08 &
INPUT... IDIM=47; JDIM=51; FINTRV=20.0; NTAPE=5; REAL PSI(IDIM,JDIM);
          IDIM=47&
          JDIM=51&
          FINTRV=2.000000E+01&
          NTAPE=5&
          REAL PSI(47,51)
INPUT... CALL READ(NTAPE,'(24F3.0/23F3.0)',PSI,1,IDIM*JDIM);
          CALL READ(NTAPE,'(24F3.0/23F3.0)',PSI,1,IDIM*JDIM)
INPUT... CALL NEWPLT('M6350','2024','VELLUM','BLACK');
          CALL NEWPLT('M6350','2024','VELLUM','BLACK')
INPUT... CALL CALTUR(PSI,IDIM,JDIM,0,0,-1,FINTRV,0,2.0,6.0,8.52,-1.0);
          CALL CALTUR(PSI,IDIM,JDIM,0,0,-1,FINTRV,0,2.0,6.0,8.52,-1.0)

 CONTOUR PROGRAM HAS BEEN ENTERED.
 MINIMUM IS  5.910000 02. MAXIMUM IS  9.490000 02. CONTOUR INTERVAL IS  2.0E 01.
 CONTOUR PROGRAM TOOK   3.31 SECONDS.
          INPUT... CALL GLOBE(2,90.0,280.0,0,4.070136,0,2.0,6.0,8.52);
          CALL GLOBE(2,90.0,280.0,0,4.070136,0,2.0,6.0,8.52)
GLOBE CALLED. USED  10.90 SECONDS.
          INPUT... CALL PLOT1(14.0,0,-3); CALL ENDPLT; END;
          CALL PLOT1(14.0,0,-3)
          CALL ENDPLT
          END
```



MAXIMUM VALUE IS 949.0
MINIMUM VALUE IS 591.0
CONTOUR INTERVAL IS 20.0

Fig. 5.1 Contour Map with geographic outlines, produced by Example 1 using Basic COMS.

```
      SUBROUTINE READ(/ITAPE/,FORMAT,ARRAY,/IMIN/,/IMAX/)
      INTEGER FORMAT(1),ARRAY(1)
      NARG=NUMP(0)
      IF(NARG.LT.3) RETURN
      IF(NARG.GT.5) RETURN
      IMN = 1
      IMX = 1
      NARG = NARG - 2
      GO TO(3,2,1),NARG
1     IMX = IMAX
2     IMN = IMIN
3     READ(ITAPE,FORMAT,END=6,ERR=6)(ARRAY(I),I=IMN,IMX)
6     RETURN
      END
```

5.3 Example 2

This example, like the preceding one, uses the CALTUR program to produce its final output. However, in this case the data to be contoured is generated in a somewhat different manner. Here the data is read, in a packed form, from a data tape containing 3652 separate fields. The data is read by a library program GETDAT which accepts up to three arguments. The first is the COMS array into which the data is to be read, and the second is the number of the field which is desired from the tape. A third argument allows specification of a FORTRAN logical tape number, but in the absence of such specification tape number one is used. When a zero is given as the number of the field desired, the tape is rewound.

A second library program called OPRATN is also utilized in this example. This program allows a specified operation (addition, subtraction, multiplication or division) to be carried out between all the elements of specified arrays, the result being placed in a third array. For this example the OPRATN program is used to subtract the first field from the second field, after those fields have been read from the tape. A contour map is then produced of a portion of this difference field.

On the following pages the sequence of statements, the contour map resulting, and the OPRATN library program are shown. The program GETDAT is not presented because its operation is similar to that of the program READ in the preceding example.

```
          STRAN INTERPRETER ON MAY 22,1969 AT 03:04:50.430 PAGE  3


          BEGIN INTERPRETING RULES.


          INPLT... REAL PSI(47,51); REAL TAU(47,51);
                   REAL PSI(47,51)
                   REAL TAU(47,51)
          INPUT... I=1; CALL GETDAT(PSI,I); I=2; CALL GETDAT(TAU,I);
                   I=18
                   CALL GETDAT(PSI,I)
                   I=28
                   CALL GETDAT(TAU,I)
          INPUT... CALL OPRATN(PSI,TAU,'-',PSI,1,2397);
                   CALL OPRATN(PSI,TAU,'-',PSI,1,2397)
          INPUT... CALL NEWPLT('M6350','2024','VELLUM','BLACK');
                   CALL NEWPLT('M6350','2024','VELLUM','BLACK')
          INPUT... CALL CALTUR(PSI,47,51,'DIFFERENCE MAP  ',-4,-18361836);
                   CALL CALTUR(PSI,47,51,'DIFFERENCE MAP  ',-4,-18361836)

DIFFERENCE MAP
MINIMUM IS -1.6384E 04. MAXIMUM IS  1.6384E 04. CONTOUR INTERVAL IS  2.5E 03.
CONTOUR PROGRAM TOOK   1.25 SECONDS.
          INPUT... CALL GETDAT(PSI,0);
                   CALL GETDAT(PSI,0)
          INPUT... CALL PLOT1(14.0,0,-3); CALL ENDPLT; END;
                   CALL PLOT1(14.0,0,-3)
                   CALL ENDPLT
                   END
```

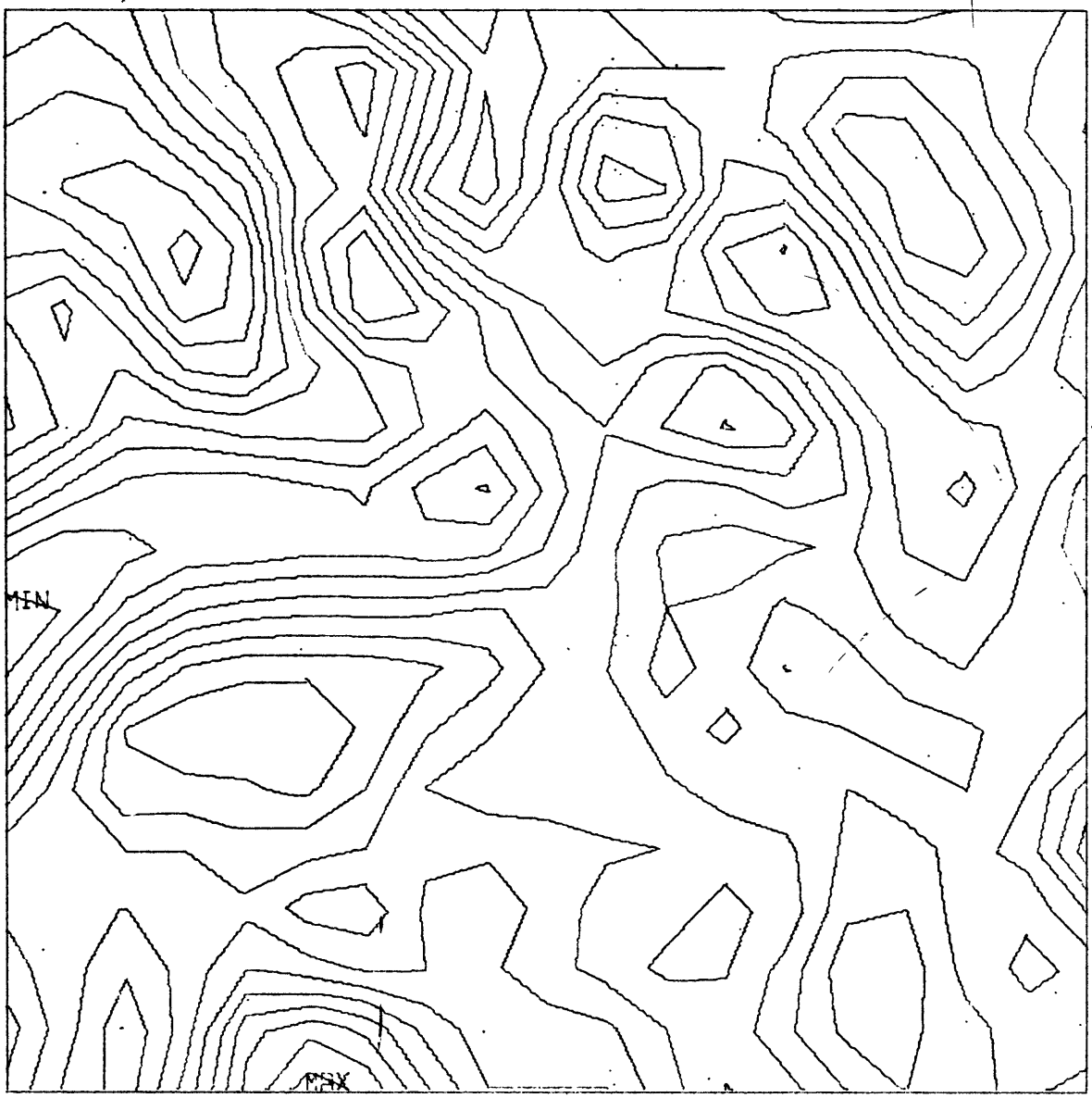


Fig. 5.2 Contour Map of the difference between two fields, produced by Example 2 using Basic COMS.

```
      SUBROUTINE OPRATN(ARRA1,ARRA2,/IOP/,ARRA3,/IBG/,/IND/)
      DIMENSION ARRA1(1),ARRA2(1),ARRA3(1)
      INTEGER IOPS(5)
      DATA IOPS/'+   ','-   ','*   ','/   ','     '/
      CALL NUMP(NARG)
      IF(NARG.LT.4) GO TO 777
      IBEG = 1
      IEND = 1
      NARG = NARG - 3
      GO TO(3,2,1),NARG
1     IEND = IND
2     IBEG = IBG
3     DO 4 IOPN=1,5
      IF(IOP.EQ.IOPS(IOPN)) GO TO 5
4     CONTINUE
      GO TO 777
5     GO TO (10,20,30,40,50),IOPN
10    DO 11 I=IBEG,IEND
11    ARRA1(I) = ARRA2(I)+ARRA3(I)
      RETURN
20    DO 21 I=IBEG,IEND
21    ARRA1(I) = ARRA2(I)-ARRA3(I)
      RETURN
30    DO 31 I=IBEG,IEND
31    ARRA1(I) = ARRA2(I)*ARRA3(I)
      RETURN
40    DO 41 I=IBEG,IEND
41    ARRA1(I) = ARRA2(I)/ARRA3(I)
      RETURN
50    DO 51 I=IBEG,IEND
51    ARRA1(I) = ARRA2(I)
      RETURN
777   WRITE(6,776)
776   FORMAT(' ERROR IN OPRATN, CONTROL RETURNED WITHOUT EXECUTION.')
      RETURN
      END
```

5.4 Example 3

In this example all of the Calcomp plots given in Appendix D, Section D.2, are generated. Those plots show the generation of a smooth curve to fit a step function, using piecewise spline interpolation.

The only new library subprogram necessary for this task is called SET. This program generates a linear function between specified indices of a specified array. The first argument of SET is the array name, the second is the initial index setting, the third is the final index setting, the fourth argument is the function value at the initial index position and the fifth is the function value at the final index position. In the absence of the second through fifth arguments, indices are assumed to take the value one, and function values default to zero. Thus, if only the array name is given, its first location will be set to zero. The listing of the SET program is given at the end of this example.

The OPRATN library program, introduced in the preceding example, is used in conjunction with the SET program to produce the functions desired. A Calcomp library program PICTUR, which generates graphs, is then used for plotting the results. Those results are presented in Appendix D.

BEGIN INTERPRETING RULES.

```
INPUT... N1=101; N2=201; N3=301;
         N1=101&
         N2=201&
         N3=301&
INPUT... REAL X(N3); REAL Y1(N3); REAL Y2(N3); REAL Y3(N3);
         REAL X(301)
         REAL Y1(301)
         REAL Y2(301)
         REAL Y3(301)
INPUT... CALL SET(X,1,N3,-1.0,2.0);
         CALL SET(X,1,N3,-1.0,2.0)
INPUT... CALL SET(Y3,1,N1,0.0,1.0); CALL SET(Y3,N1,N2,1.0,0.0);
         CALL SET(Y3,1,N1,0.0,1.0)
         CALL SET(Y3,N1,N2,1.0,0.0)
INPUT... CALL SET(Y3,N2,N3); CALL OPRATN(Y3,Y3,'*',Y3,1,N2);
         CALL SET(Y3,N2,N3)
         CALL OPRATN(Y3,Y3,'*',Y3,1,N2)
INPUT... CALL SET(Y2,1,N1,3.0,1.0); CALL SET(Y2,N1,N2,1.0,3.0);
         CALL SET(Y2,1,N1,3.0,1.0)
         CALL SET(Y2,N1,N2,1.0,3.0)
INPUT... CALL SET(Y2,N2,N3);
         CALL SET(Y2,N2,N3)
INPUT... CALL CPRATN(Y1,Y3,'*',Y2,1,N3); CALL CPRATN(Y2,Y3,'*',X,1,N3);
         CALL CPRATN(Y1,Y3,'*',Y2,1,N3)
         CALL CPRATN(Y2,Y3,'*',X,1,N3)
INPUT... CALL NEWPLT('M6350','2024','VELLUM','BLACK');
         CALL NEWPLT('M6350','2024','VELLUM','BLACK')
INPUT... CALL PLOT1(0,3.0,-3);
         CALL PLOT1(0,3.0,-3)
INPUT... CALL PICTUR(5.0,3.0,'X AXIS',6,'Y AXIS',6,X,Y1,N2,0,0,X,Y2,N2,0,0);
```

```
        CALL PICTUR(5.0,3.0,'X AXIS',6,'Y AXIS',6,X,Y1,N2,0,0,X,Y2,N2,0,0)
INPUT... CALL PLOT1(5.0,0,-3);
        CALL PLOT1(5.0,0,-3)
INPUT... REAL Y4(N3); REAL Y5(N3); REAL Y6(N3); REAL Y7(N3);
        REAL Y4(301)
        REAL Y5(301)
        REAL Y6(301)
        REAL Y7(301)
INPUT... CALL SET(Y6,1,N3); CALL SET(Y4,N2,N3);
        CALL SET(Y6,1,N3)
        CALL SET(Y4,N2,N3)
INPUT... CALL OPRATN(Y3,Y1,'+',Y6,1,N3);
        CALL OPRATN(Y3,Y1,'+',Y6,1,N3)
INPUT... CALL OPRATN(Y4,@Y1(101),'+',Y6,1,N2);
        CALL OPRATN(Y4,@Y1(101),'+',Y6,1,N2)
INPUT... CALL OPRATN(Y5,Y3,'+',Y4,1,N3);
        CALL OPRATN(Y5,Y3,'+',Y4,1,N3)
INPUT... CALL PICTUR(6.,2.,'X AXIS',6,'Y AXIS',6,X,Y3,N3,0,0,X,Y4,N3,0,0,X,Y5,N3,0,0);
        CALL PICTUR(6.,2.,'X AXIS',6,'Y AXIS',6,X,Y3,N3,0,0,X,Y4,N3,0,0,X,Y5,N3,0,0)
INPUT... CALL PLOT1(5.0,0,-3);
        CALL PLOT1(5.0,0,-3)
INPUT... CALL OPRATN(Y3,Y2,'+',Y6,1,N3);
        CALL OPRATN(Y3,Y2,'+',Y6,1,N3)
INPUT... CALL OPRATN(@Y4(101),Y2,'+',Y6,1,N2); CALL SET(Y4,1,N1);
        CALL OPRATN(@Y4(101),Y2,'+',Y6,1,N2)
        CALL SET(Y4,1,N1)
INPUT... CALL SET(Y7,1,N3,-0.5,-0.5);
        CALL SET(Y7,1,N3,-0.5,-0.5)
INPUT... CALL OPRATN(Y6,Y3,'*',Y7,1,N3);
        CALL OPRATN(Y6,Y3,'*',Y7,1,N3)
INPUT... CALL OPRATN(Y7,Y4,'*',Y7,1,N3);
        CALL OPRATN(Y7,Y4,'*',Y7,1,N3)
INPUT... CALL OPRATN(Y6,Y6,'+',Y7,1,N3);
```

```
        CALL CPRATN(Y6,Y6,'+',Y7,1,N3)
INPUT... CALL CPRATN(Y7,Y5,'+',Y6,1,N3);
        CALL CPRATN(Y7,Y5,'+',Y6,1,N3)
INPUT... CALL PICTUR(6.,2.,'X AXIS',6,'Y AXIS',6,X,Y3,N3,0,0,X,Y4,N3,0,0,X,Y6,N3,0,0);
        CALL PICTUR(6.,2.,'X AXIS',6,'Y AXIS',6,X,Y3,N3,0,0,X,Y4,N3,0,0,X,Y6,N3,0,0)
INPUT... CALL PLOT1(5.0,0,-3);
        CALL PLOT1(5.0,0,-3)
INPUT... CALL PICTUR(6.,2.,'X AXIS',6,'Y AXIS',6,X,Y5,N3,0,0,X,Y6,N3,0,0);
        CALL PICTUR(6.,2.,'X AXIS',6,'Y AXIS',6,X,Y5,N3,0,0,X,Y6,N3,0,0)
INPUT... CALL PLOT1(5.0,0,-3);
        CALL PLOT1(5.0,0,-3)
INPUT... CALL PICTUR(6.,2.,'X AXIS',6,'Y AXIS',6,X,Y7,N3,0,0);
        CALL PICTUR(6.,2.,'X AXIS',6,'Y AXIS',6,X,Y7,N3,0,0)
INPUT... CALL PLOT1(5.0,0,-3);
        CALL PLOT1(5.0,0,-3)
INPUT... CALL ENDPLT; END;
        CALL ENDPLT
        END
```

```
      SUBROUTINE SET(ARRAY,/IBG/,/IND/,/FBG/,/FND/)
      DIMENSION ARRAY(1)
      FBEG = 0.0
      FEND = 0.0
      FDELTA = 0.0
      IBEG = 1
      IEND = 1
      CALL NUMP(NARG)
      GO TO(5,4,3,2,1),NARG
1     FEND = FND
2     FBEG = FBG
3     IEND = IND
4     IBEG = IBG
5     DENOM = IEND - IBEG
      IF(DENOM.EQ.0.0) GO TO 6
      FDELTA = (FEND - FBEG)/DENOM
6     DO 7 I=IBEG,IEND
      ARRAY(I) = FBEG
7     FBEG = FBEG + FDELTA
      RETURN
      END
```

5.5 Example 4

This example is a demonstration of the fact that almost any Fortran program can be placed in the COMS library and utilized, without any necessity for modification of the program. However, when the program is a main program, as is the case here, then little obvious advantage accrues from the association with COMS. Thus, although rewriting of the program is not essential, it would be desirable in order to make it into a general purpose utility subprogram instead of a self sufficient and uncommunicative main program. All of the library programs presented earlier in this chapter were general purpose subprograms, and as a result could be used together in numerous different ways. This program, in its present form, can only be used by itself.

The program being described is a two level numerical model of the atmosphere operating in a zonal ring between latitudinal walls. It is invoked by the simple statement, "CALL MODEL;". Direction of the model is carried out by the SET-RESET method described in Appendix A. The program, once called, continues to read namelist input streams until it encounters an end-of-file, at which time it calls EXIT (returns to the system). Since normally this program was executed directly from OS360, no need was seen for it to communicate the results of its computations to a higher level. Thus, the program takes care of all of its own input and output. Inputs are namelist input streams and a formatted data deck contained initial values for the model's arrays. Outputs are the settings of all namelist variables after initialization, and sequences of contour maps (on the line printer) produced

at specified times during the time integration of the model. Those outputs for a two day run of the model are presented on the following pages. The contour maps produced show the temperature field of the model at the initial time and two days later.

The running of this model under COMS does not demonstrate any particular advantage, other than the fact that the model was not changed at all to allow the association. In fact object decks were used which had been produced under an earlier version of the Fortran compiler, which is no longer available at MIT.

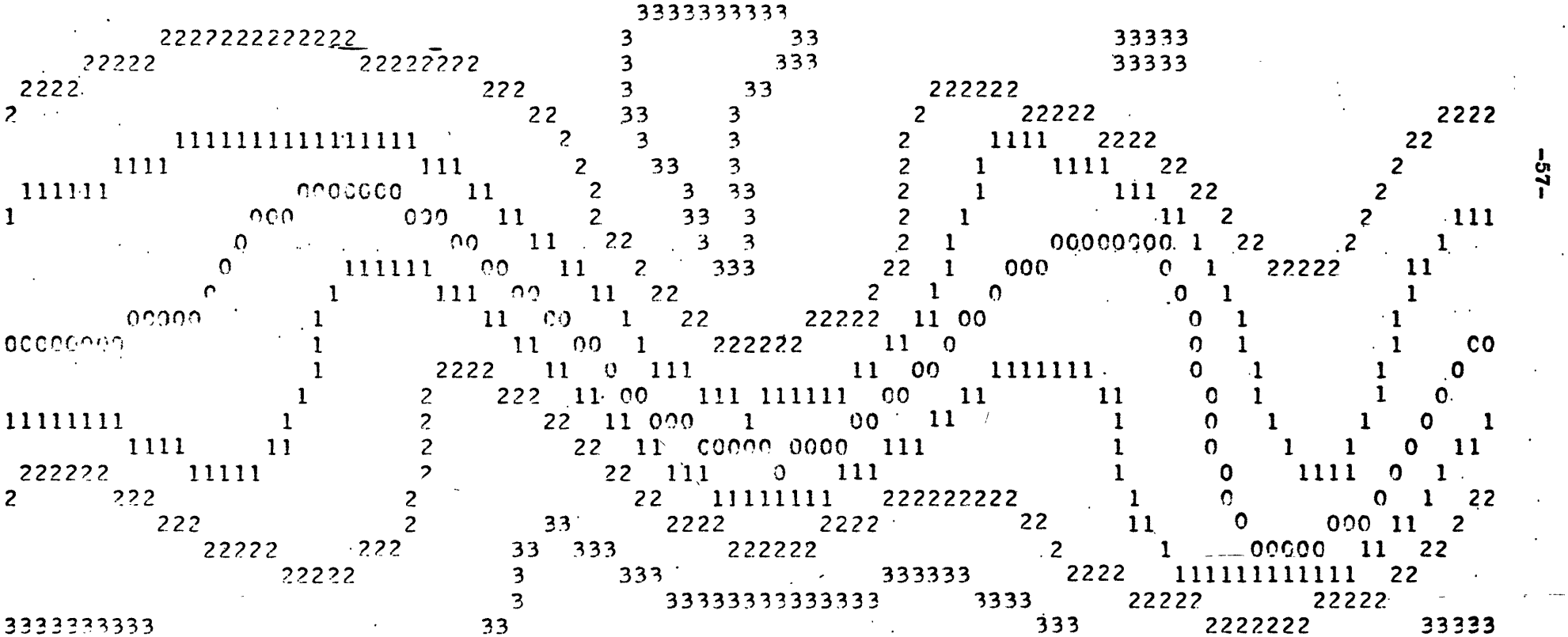
In order to improve the working relationship between COMS and this model, the most obvious approach would be to rewrite the main program of the model as a subroutine which would return to COMS after each directed session of forward time integration. The other necessity would appear to be the use of COMS arrays for representation of the fluid state. This last would allow user choice of contouring methods and user analysis of results eg. subtraction of initial field from final field as in example 2 of this chapter.

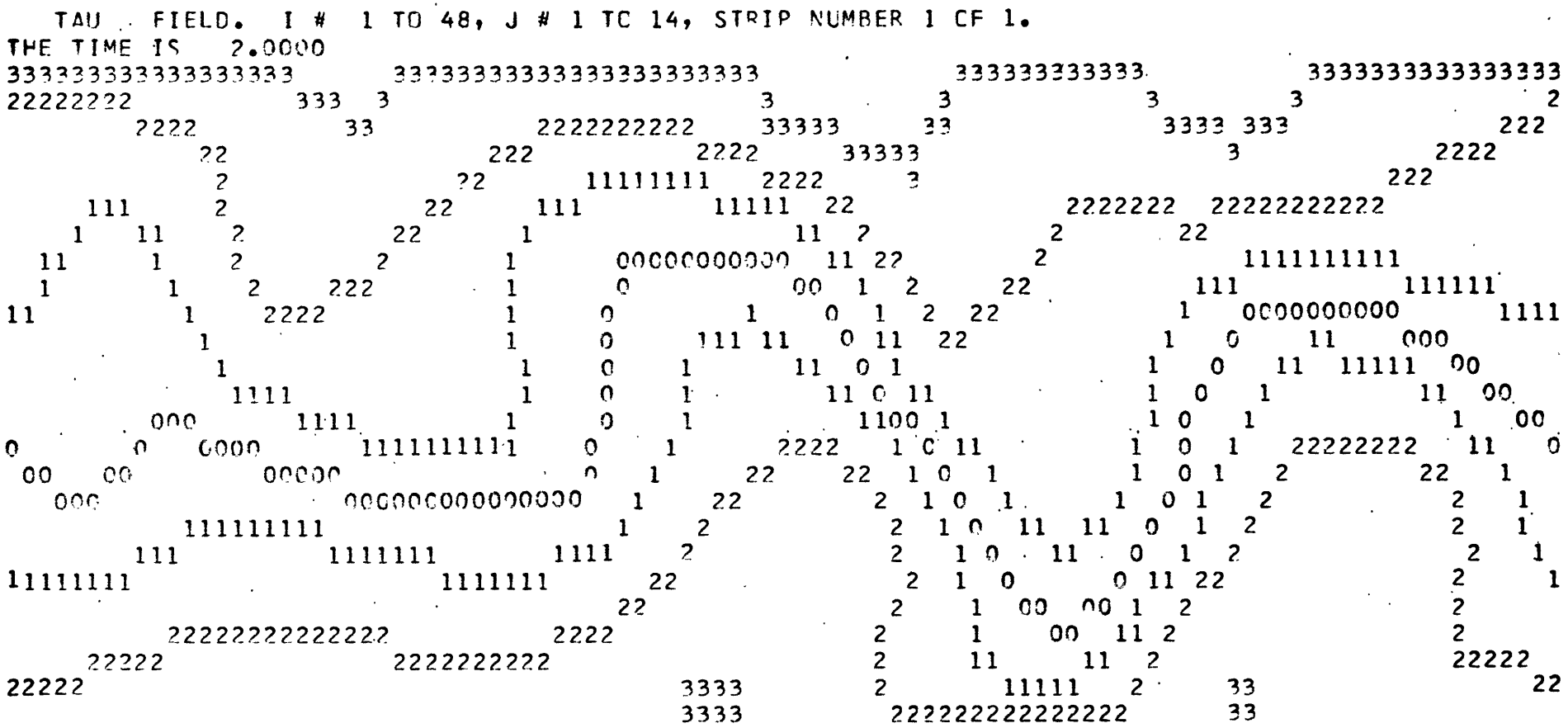
Even without making any changes in the programming of the model, a number of things could be done to make the utilization of this model from COMS more fruitful. These would involve the programming of the COMS system (in the STRAN language) to allow the storage and retrieval of namelist input streams and formatted data by user-given names. If this were done, the user could begin to cause execution of the model with statements like:

INTEGRATE CASE2 DATA FORWARD 2.0 DAYS UNDER BAROTROPIC ASSUMPTIONS.

Here CASE2 would be the name of a formatted data set, and 2.0 would be the value of the namelist variable TTOTAL, specifying the length of the time integration to be carried out. BAROTROPIC would be the name of a stored character string suitable for namelist input, to which TTOTAL = 2.0 would be appended. The resulting string would be passed to the namelist interface mechanism to be collected by the model. Thus, an inexperienced user could use the program without having to know the names of the internal control parameters and without having to provide his own initial data fields.

```
 &BEGIN
 ISIZE=          24,JSIZE=           7,TIME= 0.0           ,DELTAT= 0.49999997E-01,RUNGA=T,
 TTOTAL=  2.0000000     ,EPSILN= 0.99999990E-04,ALPHA=  1.4599991     ,LIMIT=         100,BE
   2.0000000     ,K=  2.0000000     ,KPRIME=  2.0000000     ,PPHASE= 0.0           ,TPHASE=
 TCONST= 0.0           ,NPLON=           1,NPLAT=           C,NTLON=           1,NTLAT=
 PRELAX=F,RESET=F,PRINIT=F,THETAA=-0.13999999     ,THETAB= 0.90999997     ,DTAUA= 0.0
  0.0           ,DPSIB= 0.0           ,MU=  15.000000     ,SERIES=F,PUDPSI=F,PUDTAU=F,PUTH
  0.39999998     ,CK2=  2.0000000     ,PRTHET=F,DHIST= 0.99999964E-01,HISTRY=F,TPPSI=F,TPTA

  TAU  FIELD.  I #  1 TO 48, J # 1 TO 14, STRIP NUMBER 1 OF 1.
THE TIME IS  0.0
```




TAU . FIELD. I # 1 TO 48, J # 1 TC 14, STRIP NUMBER 1 CF 1.
THE TIME IS 2.0000

CHAPTER VI

THE STRING TRANSFORMATION (STRAN) LANGUAGE AND ITS INTERPRETER

6.1 The Interpreter

The string transformation (STRAN) rule interpreter is a sequentially executed character string processor. Both the programs (rules) and data of the processor are strings of characters. Each STRAN rule can carry out matching, decomposition, transformation, and recomposition on character string data.

The three basic storage mechanisms used by the interpreter are the following:

(1) A storage for variable length strings of characters, where data and rules are kept. Each location of this storage (whether used for storage of a rule or data) is referenced by a unique name assigned at execution time.

(2) Ten special character string storage locations for the results of decompositions carried out by rules. These may be thought of as pseudo-registers or accumulators, much in the manner of assembly language programming. The pseudo-registers are referenced by the integers 1 to 10.

(3) A push-down store (similar in effect to the pop-up dish racks in cafeterias) which holds a sequence of STRAN

rule names. When the STRAN processor is asked to find and interpret a rule whose name is END it pops up the rule name at the top of the push-down and uses that instead.

The STRAN interpreter has two modes of operation. The interpreter always begins execution in "rule reading" mode. In this mode rules are collected from the input and stored, each under its own name. When the name of a rule surrounded by parentheses (eg. "(R1)") is passed to the interpreter while in this mode, control is passed to that rule, and "rule interpretation" mode begins. Return to "rule reading" mode occurs when the name of the next rule to be interpreted is END, and the push-down store is empty.

6.2 STRAN Commands

When it is in rule reading mode, the interpreter is able to accept commands from the user. These commands are comprised of reserved words surrounded by parentheses. The commands allow the user to set internal switches controlling the mode of interpreter operation or to cause data to be moved between core and secondary storage. If the character string contained within the parentheses does not match a reserved word or phrase given below, it will be treated as a rule name to receive control, as described in the preceding section. The reserved names, and the effect they cause are:

COMMANDS FOR SETTING SWITCHES

(1) (PRINT): Causes the printing of a trace of rules executed and the contents of each variable during rule execution.

(2) (NOPRINT): Printing of execution trace is discontinued.

(3) (ECHO): Causes an echo of each input card to be printed. That echo is marked by the initial phrase "INPUT..."

(4) (NOECHO): Echoing is discontinued.

(5) (PUNCH): Causes punching of a card for each output line produced by a rule.

(6) (NOPUNCH): Punching is discontinued.

The initial, or default, settings are (NOPRINT) (ECHO) (NOPUNCH).

COMMANDS TO MOVE DATA BETWEEN CORE AND SECONDARY STORE

(1) (GET RULES ddname-member): Brings to core a rule set which is a member of the data set whose ddname is given.

(2) (STORE RULES ddname-member): Stores the rule set presently in core under the given ddname and member name in secondary store.

(3) (GET MODEL ddname-member): Brings to core a collection of n-tuples for the associative memory.

(4) (STORE MODEL ddname-member): Stores the collection of n-tuples presently in core under the given ddname and member name in secondary store.

The DDNAME (an OS360 term described in the Job Control Language Manual, C28-6539-4) given by the commands above must reference a partitioned data set.

6.3 The STRAN Rule

The name of a STRAN rule and the names of rules to which it will pass control are placed at the beginning and end of the rule respectively, marked off by parentheses.

(RULE NAME(RULE BODY)NAMES OF SUCCEEDING RULES)

Either one or two names of succeeding rules may be given. If two are given, they are separated by a comma and the first will receive control if character string decomposition by the RULE BODY fails. The second name will receive control if decomposition succeeds. Examples are

(R1(body)R2)

(R2(body)R3,R4)

The rule named R1 will always pass control to the rule name R2. R2 will pass control to R3 if its body is unsuccessful at carrying out a decomposition, otherwise control will be passed to a rule named R4.

There is also a special form of rule where no names of succeeding rules are given. This form is used for putting strings of rule names

on the push-down store, and for this form the rule body must be a string of rule names separated by commas. For example:

(R5(R6,R7,R8))

The rule R5 will cause the rule names R6, R7 and R8 to be put on the push-down store with R6 on top. Since no name is given for a rule to receive control next, the top name will be collected from the push-down (leaving R7 on top), and control will be passed to R6.

All rules for the STRAN interpreter must be in one of the forms we have just described. Thus, every STRAN rule will begin with a left parenthesis, followed by the name of the rule, followed immediately by another left parenthesis. Every STRAN rule will end with two right parentheses, surrounding from zero to two rule names, which are to receive control next.

The body of a STRAN rule is where the actual processing to be carried out by the STRAN interpreter is described. The body is separated into two sides (left and right) by an equal sign. The left side of the rule body gathers the contents of storage locations named there, decomposes their contents according to pattern matching directions given, and stores the resulting decomposed elements in successive pseudo-registers. The right side of a rule body concatenates the contents of specified pseudo-registers and character literals, storing the results in the storage locations named.

An example of a simple rule will serve to motivate the descriptive material that follows.

```
(R1(VARB|$+'A'+$|=VARB|1+'B'+3|)R2,R1)
```

As can be seen the body of this rule is split into sides by an equal sign. On the left side, the body begins with the name (VARB) of a storage location followed by a vertical bar. Between that bar and the next vertical bar appear a sequence of operators separated by plus signs. Those operators will attempt to decompose the character string stored at VARB. Two different operators are shown. The dollar operator will match any arbitrary character string including the null string. The character string literal (string of characters surrounded by single quotes) will match only an exact occurrence of the contained string of characters. The operators shown above will attempt to find an A in the character string stored at VARB. If an A is found (i.e. decomposition is successful), all characters preceding the leftmost A will be placed in pseudo-register 1, the A will be placed in pseudo-register (p.r.) 2 and the remaining characters will be placed in p.r. 3. Had the decomposition been unsuccessful, control would immediately pass to a rule named R2.

Proceeding to the right side of the rule, immediately after the equal sign we find the name of a storage location (VARB) followed by a vertical bar. Between that bar and the next vertical bar appear a sequence of operators separated by plus signs. These operators will compose a character string from the contents of pseudo-registers and

character literals, the result to be stored at VARB. The 1 and 3 in this operator string refer to the contents of pseudo-registers 1 and 3. Thus, the contents of VARB will be the contents of p.r. 1 followed by a B followed by the contents of p.r. 3. The end result of a single pass through the rule is that the leftmost A in the string of characters stored at VARB will be changed to a B. Finally, because the rule passes control to itself after every successful pass, every A in the character string stored at VARB will be changed to a B. Control will be passed to the rule named R2 only when no more A's can be found.

A rule with form similar to the preceding, but carrying out the more useful task of collapsing strings of consecutive blanks, is the following:

```
(R2(VARB|$+'   '+$|=VARB|1+' '+3|)R2,R1)
```

An asterisk preceding a name on the left side of a rule body indicates that an input line is to be read into the storage location before decomposition of its contents begins. An asterisk preceding a name on the right side of a rule body indicates that after results have been placed in the storage location, its contents should be written as a line of output. An example is:

```
(READ(*INPUT|'C'+$|=*INPUT|2|)END,READ)
```

This rule will read an input line, test to see if that line begins with C, and if so it will output the rest of the line. This process will continue till an input line which does not begin with C is found.

A # sign preceding a name on the right side of a rule body indicates that before the results of composition are placed in the storage location, they should be passed to the evaluator. The result returned by the evaluator (always a string of characters) is placed in the storage location. An example is:

```
(EVAL(INPUT|$|= #INPUT|1|)END)
```

This rule will cause the whole contents of the storage location named INPUT to be passed to the evaluator. The use of this prefix is the sole means by which the interpreter may be caused to communicate with the evaluator.

A rule body need not have both right and left side. An equal sign is included only when necessary to mark the beginning of the right side. If a rule body has no left side, the contents of the pseudo-registers will be what was left from interpretation of previous rules. If more than one name and decomposition operator string appear on the left side of a rule, the operators in the separate strings will place their respective components of the decomposition in consecutive pseudo-registers. Thus, for the rule,

```
(DECOMP(VARB|$+'A'+$|TEMP|$+'B'+$|)END)
```

if VARB contains an A and TEMP contains a B, the A will appear in pseudo-register 2, and the B in pseudo-register 5 after decomposition.

A complete list of decomposition and composition operators is given in Appendix F.

6.4 STRAN and the Associative Memory

We have already described the language mechanism by which the interpreter communicates with the evaluator. Now it is time to examine the somewhat more complicated problem of communicating with the associative memory. The associative memory stores and retrieves ordered pairs, triples and quadruples of character strings. These are usually referred to as ordered n-tuples for simplicity. An n-tuple may be retrieved from the associative memory when only some of its component character strings are known. For a precise description of the processes of the associative memory, see Appendix C.

Retrieval of character strings from the associative memory, as might be expected, occurs on the left side of a STRAN rule body, with retrieved strings being placed in specified pseudo-registers. Failure to find a suitable answer or answers, like failure to decompose a character string, causes control to be passed to the rule whose name has been given for the failure case. Retrieval is carried out by two separate requests. The FIND request attempts to find suitable stored n-tuples to serve as answers for the n-tuple sought. If all components of that sought n-tuple are known, then whether or not it is stored is all the information which can be collected. However, if some components of the n-tuple were not known, the character strings to fill those voids must be collected. The ACCESS request carries out this duty. Each time an ACCESS request is interpreted, character strings are collected from specified positions in the next answer. The positions

from which strings are collected are specified by the positions of the voids in the originating FIND request.

Storing of n-tuples in the associative memory is a somewhat less complicated process. A STORE request must occur on the right side of a STRAN rule body. Strings stored as the components of the n-tuple may be literals or the contents of pseudo-registers.

The following is the form of each of the three requests mentioned above.

(1) FIND written "&Fn|pseudo-register numbers, literals and $ signs|". An example is &F5|1+$+'TED'|, which will retrieve all ordered 3-tuples whose first element is the string contained in pseudo-register 1, second element is anything and third element is the string 'TED'. A FIND request may occur only on the left side of a rule body. Failure to find any satisfactory n-tuples causes rule failure. The number immediately following the F is used in ACCESSing (see next request). That number becomes associated with the chain of answers found by this FIND request.

(2) ACCESS written "&An|pseudo-register numbers|". An example is &A5|6|, which will retrieve the character string, associated with the $ sign in the originating FIND, for the next answer on chain 5 and place it in pseudo-register 6. The access request is useful for collecting the results of FINDs which have included $ signs. Such a FIND creates a numbered pointer to a chain of answers. The positions in those answers which match up with the $ signs in the FIND

can be retrieved by accessing that chain with its number, and providing pseudo-register numbers for each of the dollar signs in the initiating FIND. One access must be made for each member of the chain of answers. If the initiating FIND contained no dollar signs then the success or failure of that FIND is the only useful information and ACCESS will not return any valid information. ACCESS requests can occur only on the left side of the rule body, and failure (no more answers on the chain) causes rule failure.

(3) STORE written "&S⌊pseudo-register numbers and literals⌋". An example is &S⌊1+'ONE'⌋, which will store an ordered pair whose first element is the character string in pseudo-register 1 and second element is 'ONE'. A STORE request may occur only on the right side of a rule body. Each element of its operator sequence serves as an element of the n-tuple to be stored. If the store request fails (this n-tuple had previously been stored) execution of the rest of the rule is skipped, but control is transferred as if the rule had succeeded.

6.5 Summary

The following is a capsule summary of the syntax of rule bodies. This summary, in conjunction with the complete list of decomposition and composition operators given in Appendix F, completes our description of the features of the STRAN language. In this summary, square brackets indicate an optional element, curvy brackets indicate a choice of one from a list.

(1) left and right side of the rule body are separated by =

(2) left side: one or more syntatic units from the following,

[*] storage name|decomposition operator string|

&Fn|string of $ signs, literals and pseudo-reg. numbers|

&An|string of pseudo-register numbers|

(3) right side: one or more syntactic units from the following,

[{*, #}] storage name|composition operator string|

&S|string of literals and pseudo-register numbers|

(4) All strings of operators between vertical bars are separated by plus signs.

As can be seen from this summary, STRAN is an extremely simple language. At the same time it is very powerful. The combination of simplicity and power, coupled with the implementation of the interpreter in PL/1, make STRAN a language that is easy to learn and easy to use. Being programmed in PL/1 also means that adding new features to the STRAN language is relatively easy.

6.6 Example STRAN Rule Sets

An example of a simple program illustrating the use of STRAN rules is the following:

```
(PROG(READ,EXEC,TEST))
(READ(*INPUT|'*'+$|=*INPUT|'COMMENT...'+2|)END,READ)
```

```
(EXEC(INPUT|$+'('+$+')'+$|=*OUTPUT|3|INPUT|5|)END,EXEC)
(TEST(INPUT|$+')'+$|)PROG,END)
```

As can be seen, the last three rules of the four operate as subroutines called by the rule PROG. This simple program will search through a sequence of input cards, printing all cards with an initial asterisk as comments, printing any character sequence enclosed in parentheses on a separate line, and terminating when an excess right parenthesis is found. The only cards which must be added to cause execution are a command "(PROG)" to cause control to be passed to the first rule, followed by a set of data cards.

Because rules and data are stored in the same mechanism the user of STRAN must take care that he never causes the processor to interpret a string which is not a well-formed rule. The lack of distinction between rules and data has its dangers, but provides possibilities of using sophisticated programming techniques, e.g. a program modifying itself. It appears possible, because of this feature and others, to write a STRAN processor in the STRAN language. Such an exercise has not been carried out, but might be of some theoretical interest at a later date when time is in greater supply.

A simple example of a use of the fact that rules and data are stored in the same mechanism is the following (notice that when a rule is stored internally the pair of left parentheses and the name they surround are removed from the front of the rule):

```
(RULE(=RULE1|'RULE2,RULE3))'|)RULE1)
```

Thus, this rule loads up the location RULE1 with a character string which makes a satisfactory rule, and then control is transferred to it. The end result is that the names RULE2 and RULE3 are pushed onto the push-down store.

Our final example of STRAN programming is the rule set which was used to produce all the example runs of Chapter V. This rule set reads input lines, separates them into statements by finding semicolons, and then passes the statements to the evaluator. The only new concept which is introduced in this example is the use of the dollar literal decomposition operator (described in Appendix F). In this case, the literal is always a single blank, so the operator appears as follows:

```
$' '
```

This operator matches an arbitrary number of consecutive occurrences of the associated literal, including none. Thus, the particular form of the operator used in this example will match zero or more blanks. This allows the easy elimination of blanks occurring where they are not desired.

This completes our description of the STRAN language and its interpreter. In Appendix B a number of examples of useful STRAN language programs are presented. Those examples go much farther toward clarifying what can be done in the STRAN language and ways of doing it.

```
STRAN INTERPRETER ON MAY 22,1969 AT 03:04:50.220 PAGE  2


BEGIN READING RULES.


INPUT...(COMPUTE(READ,EVAL))
INPUT...(READ(*INPUT|'#'+$|=*INPUT|2|)END,READ)
INPUT...(EVAL(INPUT|$+';'+$|=INPUT|3|NEXT|1|)COMPUTE,EVL1)
INPUT...(EVL1(NEXT|$' '+'END'+$' '|=*NEXT|2|)EVL2,END)
INPUT...(EVL2(NEXT|$' '+'CALL'+$|=*NEXT|2+3|#NEXT|2+3|)EVL3,EVAL)
INPUT...(EVL3(NEXT|$' '+$|=#NEXT|2|)EVL4)
INPUT...(EVL4(NEXT|$|=*NEXT|1|)EVAL)
INPUT...(COMPUTE)
```

CHAPTER VII

MODELING AND FACT RETRIEVAL

7.1 The Set Theoretic Language

In the previous chapter we described how the interpreter may store and retrieve ordered n-tuples of symbols (strings of characters) in the associative memory. Now we will examine how that memory mechanism may be used. The specific application for which the associative memory was originally designed is the storage and retrieval of sentences from finite set theory. The purpose of these sentences is to provide a "self model" for COMS. It must be emphasized that the interpretation of stored n-tuples as sentences of finite set theory encoded in a particular form, is only one of a myriad of potential uses of the associative memory or interpretations of its stored information. This particular application and interpretation appear to be extremely fruitful, however, and to merit much deeper exploration than any other that has been developed so far.

The Set Theoretic Language (STL) as described by Gammill (1966) is not a new language, per se, but simply an encoding of the sentences of finite set theory (Halmos 1960) into the form of ordered n-tuples which can be stored in our associative memory mechanism. We will frequently refer to these ordered n-tuples as "facts". The only confusion that can arise in the encoding process is that set theory includes ordered n-tuples in its syntax. However, for clarity we will always represent

an n-tuple of set theory by surrounding it with sharp brackets, and will represent n-tuples of symbols to be stored in our memory mechanism by delimiting parentheses. Thus the following examples are pairwise equivalent ($\in$ is the member relation and $\subset$ is subset):

	Set Theoretic Language	Language of Finite Set Theory
1)	(a,b)	$b \in a$
2)	(d,e,f)	$\langle e,f \rangle \in d$
3)	(g,h,j,k)	$\langle h,j,k \rangle \in g$
4)	($\subset$,j,k)	$j \subset k$ or $\langle j,k \rangle \in \subset$

Since such symbols as $\subset$ are missing from the computer character set, we will normally refer to that relation as "SUBSET OF". We will also expand the letters being used in the sentences to be somewhat more expressive symbols from the English language. These symbols will provide better cues to the user of the language as to the exact interpretation to be placed on each symbol. Thus, we repeat the four types of sentences of STL in expanded forms suitable for storage in the computer:

(1) (MAN,NORMAN)

(2) (FATHER OF,NORMAN,ERIC)

(3) (CHAIN OF&AND,GRANDFATHER OF,FATHER OF,PARENT OF)

(4) (SUBSET OF,FATHER OF,PARENT OF)

(5) (CHAIN OF&AND,SUBSET OF,SUBSET OF,SUBSET OF)

(6) (TRANSITIVE,SUBSET OF)

(7) (INVERSE OF,INVERSE OF,INVERSE OF)

This simple set of n-tuples demonstrates something that makes the Set

Theoretic Language more powerful than many other fact retrieval languages. It is simple to state properties and relations of the properties and relations in the language. That sentence may seem a little convoluted, but it states that symbols which appear in the first position of an STL sentence can also appear in non-first positions of sentences. Any symbol appearing in first position of an STL n-tuple is a property or relation. If the sentence is a 2-tuple then the symbol is a property, otherwise it is a relation. All properties and relations are sets. An example of a relation being stated on relations is (4) above which says that the relation FATHER OF, which pertains between NORMAN and ERIC, bears the SUBSET OF relation to the relation PARENT OF. Furthermore, the SUBSET OF relation bears the CHAIN OF&AND relation to itself and itself. As can be seen, we have already begun to build a complex hierarchy of interacting properties and relations on a simple domain containing only two individuals, (non-sets) NORMAN and ERIC.

The seven example STL sentences given above graphically demonstrate the hierarchy of properties, relations and individuals which can be built by any set of STL sentences. On the next page we show that hierarchy, and indicate where each symbol used fits in it.

Level 1.

PROPERTIES ON RELATIONS AND PROPERTIES	RELATIONS ON RELATIONS AND PROPERTIES
TRANSITIVE	CHAIN OF&AND SUBSET OF INVERSE OF

Notice that properties and relations at level 1 apply to each other, (n-tuples numbered 5 and 6) to symbols on the level below, (3 and 4) and can also be self applied (7). Properties and relations occurring at this level are frequently called "primitives".

Level 2.

PROPERTIES ON THE DOMAIN OF INDIVIDUALS	RELATIONS ON THE DOMAIN OF INDIVIDUALS
MAN	FATHER OF GRANDFATHER OF PARENT OF

Here the properties and relations apply only to the level below (n-tuples numbered 1 and 2).

Level 3.

DOMAIN OF INDIVIDUALS
NORMAN ERIC

These symbols can never appear in the first position of an STL n-tuple. They are <u>not</u> sets.

As can be seen, the top level of the hierarchy provides a set of "primitive" properties and relations which can be used to define the relationships occurring throughout the total associative structure. These primitive (top level) relations must each be provided with a procedural definition (STRAN rules) to allow the interpreter to produce deductions from the set of sentences (model) provided by a user. When this is done, a user can present a model to the system (associative processor for the STL language and deduction procedures for the primitive relations, all programmed in STRAN) and thence retrieve information which has not actually been entered, but is implied by the structure of his model.

This means that before the Set Theoretic Language system can really be considered complete, a complete set of algorithmically defined primitive relations and properties must be provided. This is done by writing a deduction procedure for each relation, in STRAN. The primitive relations can then be used by anyone desiring to model some problem space, to define the meanings of his special properties and relations (level 2) on that problem space. Thus, the process of model building using STL becomes one of identifying the relevant domain of individuals (level 3) and sets of meaningful relations and properties on this domain (level 2). Finally, the properties and relations (level 2) must be defined in terms of the primitive properties and relations provided by STL (level 1). Once this is done, deductions can be drawn from models of a problem space which concern relationships between the individuals of the domain.

In order to make this general discussion of modeling considerably more concrete, and perhaps more understandable, the reader should examine the model of human relationships (a highly intuitive area to most humans) presented in Appendix B, examples 5 through 7. The techniques being described here have been applied, and the deductions which resulted can be seen. The STRAN programs for the primitive relations used by STL (defined in Section 7.2) are given there.

The use of human relationships may seem an odd way to demonstrate a modeling capability which is to be used as a system tool for providing user information and control information in a complex user-oriented computer facility. However, some reflection should make it clear why this has been done. The types of relations that occur between people do not change with time, nor do they vary from region to region. They are not dependent upon what type of computer the model resides in, upon the architecture of the system programming used to run that computer, upon what user programs have been submitted to the program library, nor upon what data collections are available in the data library. All of these things will have an effect upon the form of an effective model to be used as a central repository for facts about the operation of a particular version of the COMS system. This means that any such model we present can only be suggestive of the kinds of techniques which can be used in modeling the characteristics of a computer system. Such a suggestive model is presented in Chapter VIII, but it cannot demonstrate the completeness nor the level of sophistication shown by the model of human relationships given in Appendix B.

7.2 A Set of Primitive Relations for STL

A complete set of primitive relations has not, as yet, been found. It seems clear that such a set could be found, for logicians such as W.V. Quine (1959) have examined systems of primitive functions similar to, but more powerful than, those given below and shown them to be complete.

The set of relations which are defined below have proved useful in practice. Little more can be said for them. If serious work were to be done in the area of deductive inference on STL models, considerably more labor would need to be expended upon the set of primitive relations. The bottom level predicate statements in the definitions given have been written as sentences of STL.

(1) (CHAIN OF&AND,A,B,C) means $(x)(y)((A,x,y) \equiv (\exists z)((B,x,z) \wedge (C,z,y)))$

(2) (SUBSET OF,A,B) means $(x)((A,x) \supset (B,x))$

or $(x)(y)((A,x,y) \supset (B,x,y))$

or $(x)(y)(z)((A,x,y,z) \supset (B,x,y,z))$

(3) (DISJOINT FROM,A,B) means $(x) \neg ((A,x) \wedge (B,x))$

or $(x)(y) \neg ((A,x,y) \wedge (B,x,y))$

or $(x)(y)(z) \neg ((A,x,y,z) \wedge (B,x,y,z))$

(4) (LEFT INTERSECTION OF&AND,A,B,C) means

$$(x)(y)((A,x,y) \equiv ((B,x,y) \wedge (C,x)))$$

(5) (RIGHT INTERSECTION OF&AND,A,B,C) means

$$(x)(y)((A,x,y) \equiv ((B,x,y) \wedge (C,y)))$$

(6) (INVERSE OF,A,B) means $(x)(y)((A,x,y) \equiv (B,y,x))$

(7) (INTERSECTION OF&AND,A,B,C) means $(x)((A,x) \equiv ((B,x) \wedge (C,x)))$

or $(x)(y)((A,x,y) \equiv ((B,x,y) \wedge (C,x,y)))$

or $(x)(y)(z)((A,x,y,z) \equiv ((B,x,y,z) \wedge (C,x,y,z)))$

(8) (UNION OF&AND,A,B,C) means $(x)((A,x) \equiv ((B,x) \vee (C,x)))$

or $(x)(y)((A,x,y) \equiv ((B,x,y) \vee (C,x,y)))$

or $(x)(y)(z)((A,x,y,z) \equiv ((B,x,y,z) \vee (C,x,y,z)))$

(9) (LEFT HALF OF,A,B) means $(x)((A,x) \equiv (\exists y)(B,x,y))$

(10) (RIGHT HALF OF,A,B) means $(y)((A,y) \equiv (\exists x)(B,x,y))$

(11) (MINISET OF&AND,A,B,C) means $(A,C) \wedge (x)((B,x) \supset (A,x))$

Besides the inadequacy of this set of primitive relations in a precise logical sense, the deductive procedures presented in example 7 of Appendix B do not even capture the full power of these primitive relations. For example, the following set of n-tuples,

(RIGHT HALF OF,OFFSPRING,PARENT OF)

(LEFT HALF OF,PARENT,PARENT OF)

(SUBSET OF,PARENT,OFFSPRING)

should generate an unbounded sequence of ancestors of any individual stated to have the OFFSPRING property. The deductive procedures in Appendix B do not do such a thing, because they have been restricted to generate only n-tuples in which all symbols occurring have been predefined. A chain of ancestors cannot be generated unless some

procedure is included to produce "token names" to stand for each of those ancestors. Why one would want to generate such a string of unnamed, but actual, individuals is also open to question. At any rate, it should be clear that certain kinds of deductions which depend upon the known existence of some unnamed individual cannot be carried out by the deduction procedures presently implemented.

Solutions to this, and many other problems, would need to be found if the STL language were to be used for sophisticated deductive inference on very complex models. The example presented in Appendix B shows that the mechanisms needed have been provided, and that a great deal can be done with a very simple deductive procedure.

7.3 An Example Deduction

In order to clarify the operations that occur when a STRAN rule set makes deductions from a set of stored sentences of STL, we present the following example. The simplest STL primitive relation, from the point of view of deduction, is INVERSE OF. Let us assume that the associative memory contains the following sentences:

1. (INVERSE OF,ABOVE,BELOW)
2. (ABOVE,LAMP,TABLE)

When the STRAN rules to produce deductions for INVERSE OF are entered, they seek all ordered triples which begin with the relation INVERSE OF. Sentence 1 will be found. That triple asserts that the relation ABOVE is the INVERSE OF the relation BELOW. As a result of this information

the STRAN rules seek all ordered triples which begin with the relation ABOVE. Finding sentence 2, the rule set is then able to store the following sentence as a deduction:

3. (BELOW,TABLE,LAMP)

Deductions may be carried out at many levels. For example, if the following sentence is also included in the memory,

4. (INVERSE OF,INVERSE OF,INVERSE OF)

then the following sentence will be deduced.

5. (INVERSE OF,BELOW,ABOVE)

Example 7 of Appendix B contains the rule set (named INVERSE) which carries out the operations described here.

7.4 Translation between STL and English

Early in the development of the Set Theoretic Language, it was noticed that the n-tuples produced often bore a marked resemblance to a simple declarative sentence of English. This resemblance can be increased by the inclusion of prepositions in the names of relations, and using the singular form of all nouns. Thus:

	STL	English
1)	(BOY,ERIC)	Eric is a boy.
2)	(TALL,NORMAN)	Norman is tall.
3)	(FATHER OF,NORMAN,ERIC)	Norman is the father of Eric.
4)	(MARRIED TO,NORMAN,MADELEINE)	Norman is married to Madeleine.

5) (SUBSET OF,MAN,MALE) Any man is a male.

6) (GOING FROM&TO,JOE,HARVARD,MIT) Joe is going from Harvard to MIT.

7) (CHAIN OF&AND,SUBSET OF,SUBSET OF,SUBSET OF)

Subset of is the chain of subset of and subset of.

The only difficulty one encounters in making the transformation between STL and English is knowing which words or phrases require a determiner (i.e., "a" or "the"). To solve this problem requires knowledge of the syntactic category of the word or phrase. Thus, BOY becomes "a boy" because BOY is a noun, while TALL stays the same because it is an adjective. Syntactic information as well as semantic information can be stored in the associative memory. One can write:

(NOUN,BOY)

(ADJECTIVE,TALL)

Using such information, translations can be easily carried out. It should be clear also that generation of English without that information will produce non-English (eg. "Eric is boy.").

Examples 6 and 8 in Appendix B show translations carried out by STRAN rule sets. Example 6 shows translation from English to STL and example 8 shows the reverse. Examples in the next chapter will show how each of these capabilities may be used to aid the user of COMS. As a result of this ability to give and receive information in a restricted (and hopefully easily learned) subset of English, COMS should be able to converse with inexperienced users who do not know STL.

CHAPTER VIII

STEPS TOWARD MORE SOPHISTICATED COMS IMPLEMENTATIONS

In this chapter we will examine more complicated implementations of COMS that allow a user greater flexibility and provide him with information and help in utilizing the program library. In Chapter V we showed how the evaluator and program library could be used with a minimal STRAN rule set. Now we will show how more complicated rule sets can provide the user with much stronger capabilities. We will also show how the associative memory contributes to those capabilities. It is intended that the reader should extrapolate from the examples presented here to visualize very advanced COMS systems that could be built using the foundation elements which have been presented. Development of such advanced COMS versions will require considerable experimentation with various forms of user language and conversational techniques, which are outside the scope of the present work. It is hoped that the following examples may suggest some ways in which such work should proceed.

8.1 Implementation Example 1: A DO loop processor

The following STRAN rule set allows a single (un-nested) DO loop to be processed by the STRAN interpreter. To accomplish this task, all statements between the DO statement and END OF LOOP statement are saved as data in the STRAN interpreter storage under a sequence of

generated names. When the loop is actually executed these saved statements are retrieved and passed to the evaluator one at a time. The form of the DO statement is the following:

DO VARBNAM=N1 TO N2 [BY N3]

The expression in the square brackets is optional, and if omitted the variable VARBNAM will be increased by one at the end of each loop (i.e., N3=1). N1, N2 and N3 may be any expressions which may be evaluated to produce integer values, including simple integer constants or variables. This example does not use the associative memory.

As in Chapter V, the user should note that lines beginning INPUT... are echoes of input lines read by the interpreter, with those characters added. The use of this marker allows easy distinguishability of input lines from output lines generated by rules.

BEGIN READING RULES.

```
INPUT...(COMPUTE(READ,EVAL))
INPUT...(READ(*INPUT|'#'+$|=*INPUT|2|)END,READ)
INPUT...(EVAL(INPUT|$+';'+$|=INPUT|3|NEXT|1|)COMPUTE,EVL1)
INPUT...(EVL1(NEXT|$' '+'END'+$' '|)EVL2,END)
INPUT...(EVL2(NEXT|$' '+'DO '+$' '+$+'='+$+' TO '+$|=COV8|4|DCLM|8|DOXR|'1'|)EVL3,DODO)
INPUT...(EVL3(NEXT|$' '+$|=#NEXT|2|)EVL4)
INPUT...(EVL4(NEXT|$|=*NEXT|1|)EVAL)
INPUT...(DODO(DOLM|$+' BY '+$|=DOLM|1|DOXR|3|)DO1)
INPUT...(DO1(DOLM|$|DOXR|$|=#NEXT|4+5+6|#DOLM|1|DOXR|4+5+4+'+'+2|STMT|'0'|)DO2)
INPUT...(DO2(DOLM|$|=DOLM|'''+1+'''|)DO3)
INPUT...(DO3(DOTST,EXSAVD,ENDTST,EVAL))
INPUT...(SAVE(=TMP|'=%SP'+1+'|'+'2'+'|'+')END)'|)TMP)
INPUT...(INCR(DOXR|$|=#TMP|1|)END)
INPUT...(ENDTST(DOVB|$|=#TMP|1|)ENDT1)
INPUT...(ENDT1(DCLM|$|TMP|$' '+.1+$' '|)CONTIN,END)
INPUT...(CONTIN(INCR,EXSAVD,ENDTST))
INPUT...(DOTST(STMT|$|=#STMT|'1+'+1|)DOT1)
INPUT...(DOT1(STMT|$|INPUT|$+'END OF LOOP'+$' '+';'+$|=SAV|6|ENDST|'''+1+'''|)DOT2,SAVE)
INPUT...(DOT2(STMT|$|INPUT|$|)DOT3)
INPUT...(DOT3(SAVE,READ,DOTST))
INPUT...(EXSAVD(=STMT|'0'|)EXLOOP)
INPUT...(EXLOOP(EX1,EVDO,EXTST))
INPUT...(EX1(STMT|$|=#STMT|'1+'+1|)EX2)
INPUT...(EX2(STMT|$|=TMP|'%SP'+1+'|'+'$'+'|'+'=INPUT'+'|'+'1'+'|'+')END)'|)TMP)
INPUT...(EVDO(INPUT|$' '+$+';'+$|=INPUT|4|#NEXT|2|)END,EVD1)
INPUT...(EVD1(NEXT|$|=*NEXT|1|)EVDO)
INPUT...(EXTST(ENDST|$|STMT|$' '+.1+$' '|SAV|$|=INPUT|5|)EXLOOP,END)
INPUT...(COMPUTE)
```

```
STRAN INTERPRETER ON MAY 24,1969 AT 14:36:57.320 PAGE  3


BEGIN INTERPRETING RULES.


INPUT... IDIM=4; REAL FUNC(IDIM); DO I=1 TO IDIM;
         IDIM=4&
         REAL FUNC(4)
INPUT...ARG=FLOAT(I)*2.1789;
INPUT...FUNC(I)=ARG**1.389;
INPUT...END OF LOOP; ARG=FUNC(1)-FUNC(4); END;
         ARG=2.178899E+00&
         FUNC(1)=2.949927E+00&
         ARG=4.357799E+00&
         FUNC(2)=7.725769E+00&
         ARG=6.536699E+00&
         FUNC(3)=1.356851E+01&
         ARG=8.715599E+00&
         FUNC(4)=2.023356E+01&
         ARG=-1.728363E+01&
```

8.2 Implementation Example 2: A command processor and system self-model.

This example will be the final, and most complete, implementation of COMS to be presented. Basic methods which are applied in this implementation of COMS have all been drawn from the implementation examples presented in Appendix B and the preceding section. This COMS brings all of those methods together, in one unified rule set, to allow much greater flexibility for the user. This COMS allows the processing of user defined commands under the influence of a user supplemented system self-model.

The self-model contains descriptions of programs available in the library, commands that can be decoded and names of grammar rules to be used to generate sequences of evaluatable statements as a result of a command. The self-model also contains descriptive material about output devices, how they relate to particular programs in the library, and specifications for particular data types which have been named. The following pages show the actual self-model used in the application examples which follow. The declarative statements of the model are presented here as sentences of the Set Theoretic Language, for brevity. Those sentences can be given in English, as was indicated in Section 7.4.

```
(PROGRAM,MODEL)
(PROGRAM,GETDAT)
(PROGRAM,SET)
(PROGRAM,OPRATN)
(PROGRAM,CALTUR)
(PROGRAM,GLOBE)
(PROGRAM,PLOT3D)
(PROGRAM,PICTUR)
(PROGRAM,CONTUR)
(PROGRAM,NEWPLT)
(PROGRAM,ENDPLT)
(PROGRAM,READ)
(SUBSET OF,CONTOUR MAP,GRAPHICAL OUTPUT)
(SUBSET OF,GEOGRAPHIC OUTLINE MAP,GRAPHICAL OUTPUT)
(SUBSET OF,GRAPH,GRAPHICAL OUTPUT)
(SUBSET OF,JNWP_GRID,TWO_DIMENSIONAL)
(SUBSET OF,JNWP_GRID,ARRAY)
(SUBSET OF,JNWP_GRID,REAL)
(MODE,INTEGER)
(MODE,REAL)
(TYPE,JNWP_GRID)
(FUNCTION,FIRST DIMENSION OF)
(FIRST DIMENSION OF,47,JNWP_GRID)
(SECOND DIMENSION OF,51,JNWP_GRID)
(CHAIN OF&AND,GOES ON,SUBSET OF,GOES ON)
(PRODUCES&ON,CALCOMP CONTOUR,CONTOUR MAP,CALCOMP PLOTTER)
(PRODUCES&ON,LINE PRINTER CONTOUR,CONTOUR MAP,LINE PRINTER)
(PRODUCES&ON,MAP,GEOGRAPHIC OUTLINE MAP,CALCOMP PLOTTER)
(COMMAND FOR,PRODUCE CONTOURS OF,CONTOUR)
(COMMAND FOR,PRODUCE CONTOURS FOR,CONTOUR)
```

```
(COMMAND,CONTOUR)
(COMMAND,LINE PRINTER CONTOUR)
(COMMAND,CALCOMP CONTOUR)
(COMMAND,PRODUCE CONTOURS OF)
(COMMAND,PRODUCE CONTOURS FOR)
(COMMAND,OPEN THE CALCOMP PLOTTER)
(COMMAND,CLOSE THE CALCOMP PLOTTER)
(COMMAND,GET DATA FOR)
(RESULT OF,CONTOUR MAP,CONTOUR)
(PROGRAM FOR,CONTUR,LINE PRINTER CONTOUR)
(PROGRAM FOR,CALTUR,CALCOMP CONTOUR)
(PROGRAM FOR,GLOBE,MAP)
(PROGRAM FOR,READ,GET DATA FOR)
(PROGRAM FOR,ENDPLT,CLOSE THE CALCOMP PLOTTER)
(PROGRAM FOR,NEWPLT,OPEN THE CALCOMP PLOTTER)
(GRAMMAR RULE FOR,%SIMPCAL,ENDPLT)
(GRAMMAR RULE FOR,%CALOPN,NEWPLT)
(GRAMMAR RULE FOR,%CALCON;1,CONTUR)
(GRAMMAR RULE FOR,%CALCOM;1,CALTUR)
(GRAMMAR RULE FOR,%READ;1,READ)
(GRAMMAR RULE FOR,%GLOBE,GLOBE)
(GRAMMAR RULE FOR&ON,%CALCON;2,CONTUR,JNWP_GRID)
(GRAMMAR RULE FOR&ON,%CALCOG;2,CALTUR,JNWP_GRID)
(GRAMMAR RULE FOR&ON,%READ;2,READ,JNWP_GRID)
```

An essential point which should be made about the self model which is being used in this implementation of COMS is that not only does the system use the model to maintain correct operation, but inexperienced users can retrieve needed information directly from it. Thus, a user not knowing exactly what commands are available to him could request that information with the STL statement (COMMAND, $), which means "list all things which are commands". The user could also ask if a particular statement is a legal command eg. (COMMAND, PRODUCE CONTOURS FOR) without actually having to try the statement to see if an error message occurs. This sort of interaction with the model will, of course, require a time-shared environment to be useful, but it is clear that the system self-model can serve double duty when that environment is available. A great help in the user inquiry process, when that situation arises, would be rules for parsing user questions written in a subset of English. On the following page we present a few inquiries which were successfully processed to retrieve information from the model. The results retrieved are printed in English, using the rule set presented in example 8 of Appendix B.

```
BEGIN INTERPRETING RULES.

INPUT...(COMMAND,$)(PROGRAM,$)(SUBSET OF,$,$))
        (COMMAND,$) HAS THESE ANSWERS:
        *CONTOUR IS A COMMAND.
        *GET DATA FOR IS A COMMAND.
        *CLOSE THE CALCOMP PLOTTER IS A COMMAND.
        *OPEN THE CALCOMP PLOTTER IS A COMMAND.
        *PRODUCE CONTOURS FOR IS A COMMAND.
        *PRODUCE CONTOURS OF IS A COMMAND.
        *CALCOMP CONTOUR IS A COMMAND.
        *LINE PRINTER CONTOUR IS A COMMAND.
        (PROGRAM,$) HAS THESE ANSWERS:
        *MODEL IS A PROGRAM.
        *READ IS A PROGRAM.
        *ENDPLT IS A PROGRAM.
        *NEWPLT IS A PROGRAM.
        *CONTUR IS A PROGRAM.
        *PICTUR IS A PROGRAM.
        *PLOT3D IS A PROGRAM.
        *GLOBE IS A PROGRAM.
        *CALTUR IS A PROGRAM.
        *OPRATN IS A PROGRAM.
        *SET IS A PROGRAM.
        *GETDAT IS A PROGRAM.
        (SUBSET OF,$,$) HAS THESE ANSWERS:
        *ANY CONTOUR MAP IS A GRAPHICAL OUTPUT.
        *ANY JNWP_GRID IS REAL.
        *ANY JNWP_GRID IS A ARRAY.
        *ANY JNWP_GRID IS TWO_DIMENSIONAL.
        *ANY GRAPH IS A GRAPHICAL OUTPUT.
        *ANY GEOGRAPHIC OUTLINE MAP IS A GRAPHICAL OUTPUT.
```

Another type of information besides the self-model resides in the associative memory mechanism of this implementation. That information is a deterministic context free grammar for generating evaluatable statements (eg. call statements to cause execution of library programs). The form of this grammar, and the rule set used to generate statements from it, are nearly identical to that presented as example 4 of Appendix B. However, in this case when multiple choices of subsidiary grammar rules are provided, the choice is specified by a number from a list provided with the initiating grammar rule name. For example, "%CALCOM;2" indicates alternative 2 is to be chosen in the generation initiated by the name %CALCOM. In example 4 of Appendix B such choices are made by a random number generator. Points of interest in this generative grammar are the fact that any non-terminal symbol may serve as the starting point of a generation, and that certain initial characters of such symbols will cause insertion of a character string from STRAN storage or the associative memory. This allows command arguments to be placed in the generated string of characters. The actual grammar used in the application examples to follow, is presented in Appendix F, along with some explanation about the form of its rules and symbols.

At this point we are ready to discuss the steps which take place in a user problem solving session using this version of COMS. The first thing the user must do is to bring the model and the rule set into core. This may be accomplished either by commands to the interpreter to retrieve the appropriate system data sets (shown in Sec-

tion 6.2), or by actually reading the model and rules in as input. The rules of this implementation will not be presented here, as the total collection is rather large. Those rules are presented in Appendix F for readers interested in examining this implementation in detail. Once the rules and model are available, control must be passed to the initial rule by the statement (EVALUATE). At this point the user can begin making statements which will carry out the intended task. This is the point at which all the following application examples begin.

The execution of this version of COMS has five phases, which will be examined below. These are:

(1) Collection of user declarations.

(2) Deduction of facts from declarations combined with the model.

(3) Initialization of output devices and allocation of space for arrays.

(4) Processing of user commands (imperative statements).

(5) Closing of output devices.

It will be important that reader bear these phases in mind in reading the output of the application examples of the following section. Each of the phases is implemented by a separate STRAN rule set.

The first phase of execution has the user supplementing the information in the system model, by making declarative statements in either simple English, the Set Theoretic Language or both. These

declarative statements will normally be concerned with the properties of outputs and data objects which are relevant to the specific run. However, more sophisticated users may, at this time, make declarations which have general applicability. Also at this time, the sophisticated user may wish to add supplementary grammar rules. Demonstrations of such general supplementary declarations will be presented in application examples 3 and 4.

Collection of declarative statements is terminated by an excess right parenthesis. After termination the STRAN rules to carry out deductive inference (demonstrated in example 7 of Appendix B) are entered, to produce deductions from the declarative statements added by the user combined with the system model. Following the deduction procedures, an initialization procedure is entered. This set of rules opens the Calcomp plotter tape, if graphical output is to be produced on that device. It also generates statements to cause the evaluator to allocate space for all arrays mentioned in the declarative statements.

Following initialization, the set of rules for interpreting commands and other imperative statements is entered. Commands are translated into sequences of calls to library programs, under the influence of the facts in the system model. Those facts can cause entirely different programs to be called by the same command, as will be demonstrated in the application examples. This rule set also contains the do-loop capability shown in Section 8.1, and can be used to pass simple statements directly to the evaluator as in the basic

COMS implementation demonstrated in Chapter V. The command interpreting rule set is terminated by an input END statement. That termination causes entry to a closing rule set, which generates a call to a library program to finalize the Calcomp tape, if graphical output is being generated for that device.

In the following section we present a number of application examples, demonstrating the capabilities of the model, grammar and rule sets which have been described in this section.

8.3 Application Examples

The first example, on a following page, shows how simply a data deck may be read and passed to a contouring program to be presented on the line printer. Only one declaration and two commands are required. The declaration states that the array PSI is a JNWP_GRID, a data type whose specifications are described in the COMS self model. The two commands "GET DATA FOR PSI" and "PRODUCE CONTOURS FOR PSI" cause the reading of data cards and the production of contours. These same two commands will appear in every one of the five examples to be presented in this section, and they will be the only commands used. The object of this exercise will be to show exactly how much power the COMS self model gives the user. In each of these five examples the user will make different declarations to be added to the model, and the result will be that the same two commands will produce five completely different sets of results. Those differing results will be comprised from three different types to be declared for PSI and

two destinations to be declared for graphical output (the contour map). In the first example the contour map has been placed on the line printer by default, since no destination for graphical output was declared. That contour map could not be presented here because of its large size. Fig. D.19 in Appendix D shows a section from that map.

Since the meanings of many of the statements appearing in the printout may be unclear, we present here an account of what can be seen to be happening. The initial phase of execution involves the collection of user declarations. An initial rule of the set with that responsibility prints out the heading indicating that following inputs will be declarations. Only one input declaration is read. That declaration is in English, so it is parsed to STL and stored in the associative memory. Both the before (English) and after (STL) versions of the declaration are printed by the parsing rules. No more declarations are found, but an excess right parenthesis is found, terminating the reading of declarations. COMS then enters its deductive phase. Using three statements from the COMS model

(SUBSET OF,JNWP_GRID,REAL)

(SUBSET OF,JNWP_GRID,ARRAY)

(SUBSET OF,JNWP_GRID,TWO_DIMENSIONAL)

and the input declaration (JNWP_GRID,PSI) the STRAN deductive rules for the primitive relation SUBSET OF produce the three deductions shown in the output. Those are the only deductions which can be produced, so control is passed to the initialization rule set. That rule set uses

the deduced information, and facts about dimensions of JNWP_GRIDs in the model, to generate the statement REAL PSI(47,51), which it passes to the evaluator. The evaluator then allocates space for the array PSI. That completes the initialization phase, so control is passed to the command processing rule set. This rule set prints a heading indicating it will be collecting input imperative (command) statements. It then proceeds to collect the next input line, which contains the two commands. Each of those commands is then expanded, using the model and grammar, to produce CALL statements which are passed to the evaluator. The exact manner in which the model is utilized in the command expansion process will be shown in the next example. The use of the context free generative grammar is clarified in Appendix F.

```
N INTERPRETER ON MAY 24,1969 AT 13:34:26.120 PAGE 15


N INTERPRETING RULES.


     FOLLOWING INPUT DECLARATIONS PROCESSED.

T...PSI IS A JNWP_GRID.              )))))

     PSI IS A JNWP_GRID.
     (JNWP_GRID,PSI)

     FOLLOWING ARE DEDUCTIONS FROM DECLARATIONS.
      (REAL,PSI)
      (ARRAY,PSI)
      (TWO_DIMENSIONAL,PSI)

     FOLLOWING INITIALIZATION STATEMENTS GENERATED.
     REAL PSI(47,51)

     FOLLOWING INPUT IMPERATIVE STATEMENTS PROCESSED.
T...GET DATA FOR PSI; PRODUCE CONTOURS FOR PSI; END;

     FOLLOWING IS EXPANSION OF COMMAND...GET DATA FOR PSI
     CALL READ(5,'(24F3.0/23F3.0)',PSI,1,2397)

     FOLLOWING IS EXPANSION OF COMMAND... PRODUCE CONTOURS FOR PSI
     CALL CONTUR(PSI,47,51,0,0,-1)
```

The input for the second example differs from that of the first only in that three new declarative statements have been added. These declarations state that graphical output goes on the Calcomp plotter, and give the user's problem number and programmer number. As a result of the statement concerning graphical output (the numbers are only used for labeling the Calcomp plot so that it will not be lost) much different actions take place. The overt differences can be observed in the output. Those differences include calls to NEWPLT and ENDPLT to open and close the Calcomp plotter tape, calling CALTUR instead of CONTUR, a call to the GLOBE program to overlay geographic outlines and a call to PLOT1 to move the plotter pen away from the plotted region on completion. The necessity for all of these differences at the call statement level would certainly not be known to a beginning user. Thus the COMS self-model makes it possible for him to move his output from one device to another without having to learn all the idiosyncracies of each device and the programs which produce output on it. The contour map which resulted from the example is presented on a following page. Now it is time to carry out a step by step examination of the use of the system model to show how the rules of the command processor use that information to produce so many differences at the CALL statement level from a single additional declaration.

Contained in the COMS self-model, before the user adds any declarations of his own, are the following two STL sentences:

1) (CHAIN OF&AND, GOES ON, SUBSET OF, GOES ON)

2) (SUBSET OF, CONTOUR MAP, GRAPHICAL OUTPUT)

The crucial assertion the user makes to supplement the self-model is:

3) (GOES ON, GRAPHICAL OUTPUT, CALCOMP PLOTTER)

With this sentence added, the definition of CHAIN OF&AND given in Section 7.2 (1), and implemented as a STRAN rule set, allows COMS to draw the following deduction, adding it to the model.

4) (GOES ON, CONTOUR MAP, CALCOMP PLOTTER)

Notice that this deduction was indeed produced, as shown by the output for the example.

The self model also contains the following STL sentences:

5) (COMMAND, PRODUCE CONTOURS FOR)

6) (COMMAND FOR, PRODUCE CONTOURS FOR, CONTOUR)

7) (RESULT OF, CONTOUR MAP, CONTOUR)

8) (PRODUCES&ON, CALCOMP CONTOUR, CONTOUR MAP, CALCOMP PLOTTER)

9) (PROGRAM FOR, CALTUR, CALCOMP CONTOUR)

10) (GRAMMAR RULE FOR&ON, %CALCOG; 2, CALTUR, JNWP_GRID)

The user then supplements these sentences with

11) (JNWP_GRID, PSI)

Now, when the user has read some data into the array PSI, he gives the command:

PRODUCE CONTOURS FOR PSI

The beginning portion of this command is recognized as a command because of sentence 5 above. The command is immediately translated to the simpler form, CONTOUR, due to sentence 6. At this point COMS seeks a program name associated with the command, by asking for answers to the question (PROGRAM FOR,$,CONTOUR). No satisfactory answers can be found. Trying another tack, COMS seeks a RESULT OF the command, and discovers (sentence 7) that it should produce a contour map. The deduction that was produced earlier (sentence 4) then tells COMS that contour maps go on the calcomp plotter, and sentence 8 points out that the command which produces contour maps on the calcomp plotter is CALCOMP CONTOUR. COMS now has a new command to check for an associated program name, and sentence 9 tells it that the appropriate program is called CALTUR.

Now having found a program to call, COMS seeks relevant information about the argument, PSI. Finding sentences 10 and 11, it discovers that PSI has a special declared type, which necessitates generation of a particular calling sequence. The grammar rule name which causes that generation is %CALCOG;2 (see Appendix F for information about generation of statements by grammar rules).

In this example the user has added two supplementary declarations (3 and 11) to the COMS self-model. Through deductive capabilities sentence 4 has been added. As a result, the command PRODUCE CONTOURS FOR PSI has been translated in a manner which utilizes stored information about the destination of contour maps and about the type of the

data collection to be contoured. If the user had not added the supplementary declarations (3 and 11), translation of the command would have proceeded much differently, as can be seen from the preceding example.

```
STRAN INTERPRETER ON MAY 24,1969 AT 14:13:49.400 PAGE  9

BEGIN INTERPRETING RULES.

        FOLLOWING INPUT DECLARATIONS PROCESSED.

INPUT...GRAPHICAL OUTPUT GOES ON THE CALCOMP PLOTTER.

        GRAPHICAL OUTPUT GOES ON THE CALCOMP PLOTTER.
        (GOES ON,GRAPHICAL OUTPUT,CALCOMP PLOTTER)
INPUT...M6350 IS THE PROBLEM_NUMBER.  2024 IS THE PROGRAMMER_NUMBER.

        M6350 IS THE PROBLEM_NUMBER.
        (FUNCTION,PROBLEM_NUMBER)
        (PROBLEM_NUMBER,M6350)

        2024 IS THE PROGRAMMER_NUMBER.
        (FUNCTION,PROGRAMMER_NUMBER)
        (PROGRAMMER_NUMBER,2024)
INPUT...PSI IS A JNWP_GRID.           )))))

        PSI IS A JNWP_GRID.
        (JNWP_GRID,PSI)

        FOLLOWING ARE DEDUCTIONS FROM DECLARATIONS.
         (GOES ON,CONTOUR MAP,CALCOMP PLOTTER)
         (GOES ON,GRAPH,CALCOMP PLOTTER)
         (GOES ON,GEOGRAPHIC OUTLINE MAP,CALCOMP PLOTTER)
         (REAL,PSI)
         (ARRAY,PSI)
         (TWO_DIMENSIONAL,PSI)

        FOLLOWING INITIALIZATION STATEMENTS GENERATED.
```

```
                FOLLOWING IS EXPANSION OF COMMAND...OPEN THE CALCOMP PLOTTER
                CALL NEWPLT('M6350','2024','VELLUM','BLACK')
                REAL PSI(47,51)

                FOLLOWING INPUT IMPERATIVE STATEMENTS PROCESSED.
          INPUT...GET DATA FOR PSI; PRODUCE CONTOURS FOR PSI; END;

                FOLLOWING IS EXPANSION OF COMMAND...GET DATA FOR PSI
                CALL READ(5,'(24F3.0/23F3.0)',PSI,1,2397)

                FOLLOWING IS EXPANSION OF COMMAND... PRODUCE CONTOURS FOR PSI
                CALL CALTUR(PSI,47,51,0,0,-1)

 CONTOUR PROGRAM HAS BEEN ENTERED.
 MINIMUM IS  5.9100E 02. MAXIMUM IS  9.4900E 02. CONTOUR INTERVAL IS  2.5E 01.
 CONTOUR PROGRAM TOOK    2.72 SECONDS.
                CALL GLOBE(2,90.,280.,0,4.070136,0,2.,6.,8.52)
GLOBE CALLED. USED  10.86 SECONDS.
                CALL PLOT1(14.0,0,-3)
                END
                CALL ENDPLT
```

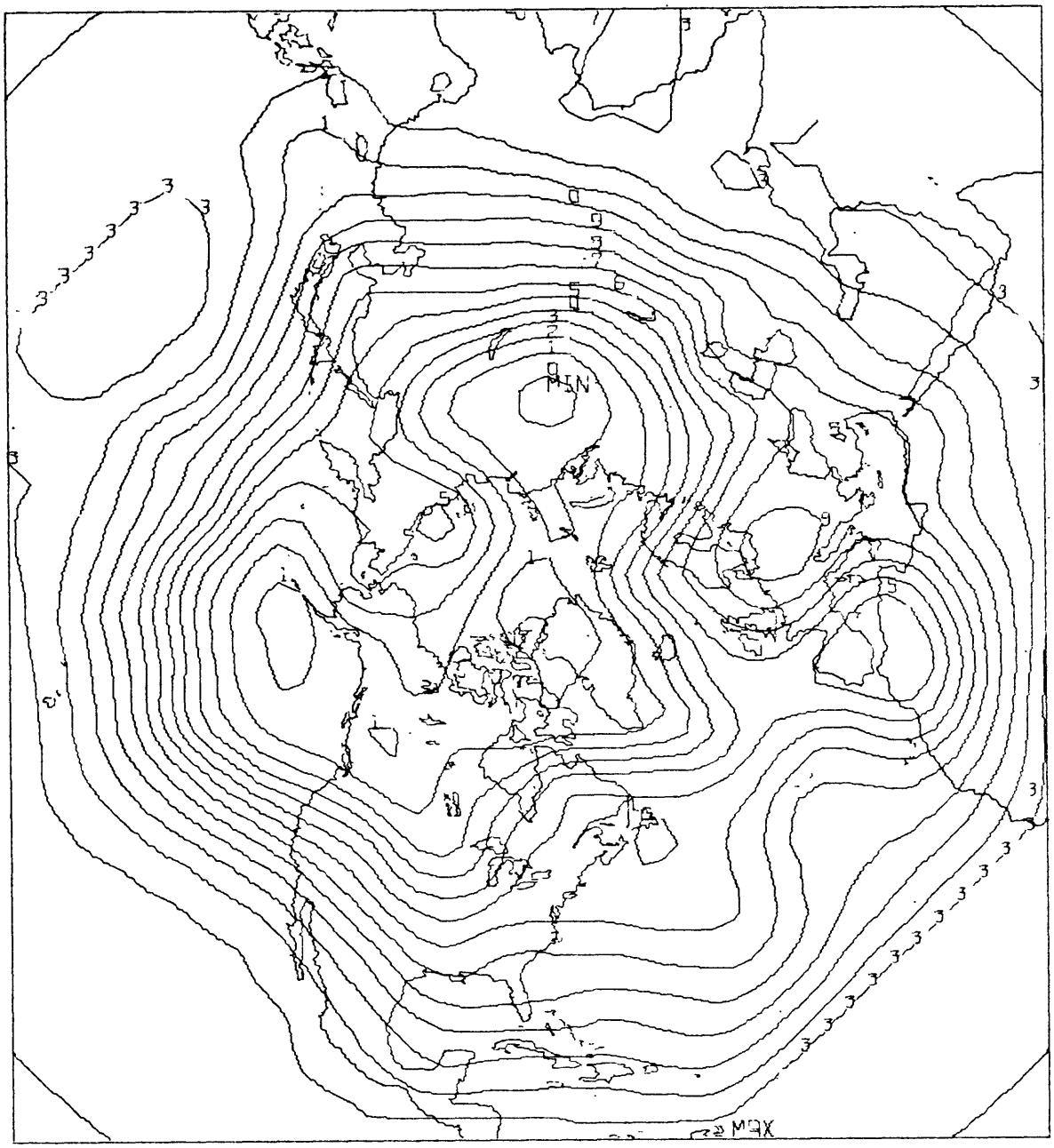


Fig. 8.1 Contour map with geographic outlines, produced by Application Example 2 of sophisticated COMS.

The third example shows how a sophisticated user may add a collection of declarative statements and grammar rules to the model, and allow himself to use a new data type. All but the last input declaration of this example are concerned with giving the specifications of that new data type, DEMO_GRID. These declarations supplement the information about what processes should take place when the same two commands (given in all our examples) occur. They tell exactly what should happen when those commands are applied to data which is declared to have the type DEMO_GRID. The contour map is placed on the line printer by default.

The input of the fourth example is identical to the third, except that the three declarations which cause graphical output to be put on the Calcomp plotter have been added. The result is that all the special calls for operating that device are generated, and the contour map is produced there. This was possible because the supplementary declarations included grammar rule information allowing correct generation of calling sequences for the program CALTUR when applied to DEMO_GRID data. Note that a DEMO_GRID, unlike a JNWP_GRID, does not have geographic outlines overlaid when it is put on the Calcomp plotter.

```
STRAN INTERPRETER ON MAY 4,1969 AT 10:06:35.47C PAGE 19


BEGIN INTERPRETING RULES.


        FOLLOWING INPUT DECLARATIONS PROCESSED.

INPUT...DEMO_GRID IS A TYPE.

        DEMO_GRID IS A TYPE.
        (TYPE,DEMO_GRID)
INPUT...*ANY DEMO_GRID IS AN ARRAY.  *ANY DEMO_GRID IS REAL.

        *ANY DEMO_GRID IS AN ARRAY.
        (NOUN,DEMO_GRID)
        (SUBSET OF,DEMO_GRID,ARRAY)

        *ANY DEMO_GRID IS REAL.
        (SUBSET OF,DEMO_GRID,REAL)
INPUT...4 IS THE FIRST DIMENSION OF A DEMO_GRID.

        4 IS THE FIRST DIMENSION OF A DEMO_GRID.
        (FIRST DIMENSION OF,4,DEMO_GRID)
INPUT...4 IS THE SECOND DIMENSION OF A DEMO_GRID.

        4 IS THE SECOND DIMENSION OF A DEMO_GRID.
        (SECOND DIMENSION OF,4,DEMO_GRID)
INPUT...%CALCOM;4 IS THE GRAMMAR RULE FOR CALTUR ON A DEMO_GRID.

        %CALCOM;4 IS THE GRAMMAR RULE FOR CALTUR ON A DEMO_GRID.
        (GRAMMAR RULE FOR&ON,%CALCOM;4,CALTUR,DEMO_GRID)
INPUT...%CALCON;3 IS THE GRAMMAR RULE FOR CONTUR ON A DEMO_GRID.

        %CALCON;3 IS THE GRAMMAR RULE FOR CONTUR ON A DEMO_GRID.
        (GRAMMAR RULE FOR&ON,%CALCON;3,CONTUR,DEMO_GRID)
INPUT...(GRAMMAR)
```

```
STRAN INTERPRETER ON MAY 4,1969 AT 10:06:38.230 PAGE 20


        FOLLOWING INPUT STORED AS GRAMMAR RULES.
INPUT...(%AL2:3=<,4,4,0,0,1,,50.0,1,4>)
INPUT...(%CALCOM=%CALCON+%MOVE)(%AL2:4=<,4,4>))
        END OF GRAMMAR.

INPUT...PSI IS A DEMO_GRID.

        PSI IS A DEMO_GRID.
        (DEMO_GRID,PSI)
INPUT...))))

        FOLLOWING ARE DEDUCTIONS FROM DECLARATIONS.
         (REAL,PSI)
         (ARRAY,PSI)

        FOLLOWING INITIALIZATION STATEMENTS GENERATED.
        REAL PSI(4,4)

        FOLLOWING INPUT IMPERATIVE STATEMENTS PROCESSED.
INPUT...GET DATA FOR PSI; PRODUCE CONTOURS FOR PSI; END;

        FOLLOWING IS EXPANSION OF COMMAND...GET DATA FOR PSI
        CALL READ(5,'(24F3.0/23F3.0)',PSI,1,IDIM*JDIM)

        FOLLOWING IS EXPANSION OF COMMAND... PRODUCE CONTOURS FOR PSI
        CALL CONTUR(PSI,4,4,0,0,1,,50.0,1,4)
```

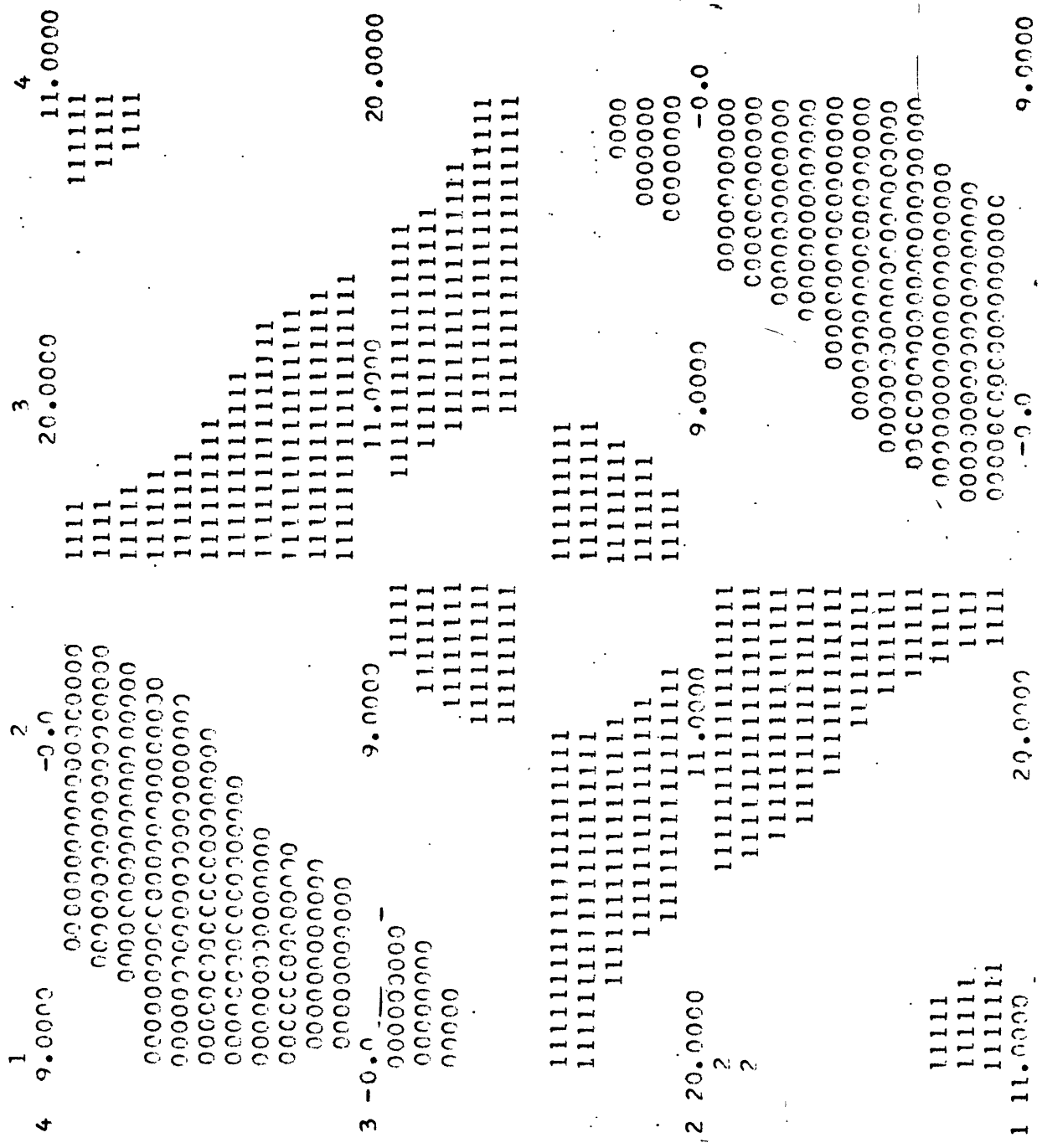


Fig. 8.2 Line printer contour map of a DEMO_GRID, produced by Application Example 3.

```
STRAN INTERPRETER ON MAY 24,1969 AT 14:48:52.210 PAGE  8


BEGIN INTERPRETING RULES.

         FOLLOWING INPUT DECLARATIONS PROCESSED.

INPUT...DEMO_GRID IS A TYPE.

         DEMO_GRID IS A TYPE.
         (NOUN,TYPE)
         (TYPE,DEMO_GRID)
INPUT...*ANY DEMO_GRID IS AN ARRAY.  *ANY DEMO_GRID IS REAL.

         *ANY DEMO_GRID IS AN ARRAY.
         (NOUN,DEMO_GRID)
         (NOUN,ARRAY)
         (SUBSET OF,DEMO_GRID,ARRAY)

         *ANY DEMO_GRID IS REAL.
         (ADJECTIVE,REAL)
         (SUBSET OF,DEMO_GRID,REAL)
INPUT...4 IS THE FIRST DIMENSION OF A DEMO_GRID.

         4 IS THE FIRST DIMENSION OF A DEMO_GRID.
         (NOUN ROOT,FIRST DIMENSION OF)
         (FIRST DIMENSION OF,4,DEMO_GRID)
INPUT...4 IS THE SECOND DIMENSION OF A DEMO_GRID.

         4 IS THE SECOND DIMENSION OF A DEMO_GRID.
         (FUNCTION,SECOND DIMENSION OF)
         (NOUN ROOT,SECOND DIMENSION OF)
         (SECOND DIMENSION OF,4,DEMO_GRID)
INPUT...%CALCOM;4 IS THE GRAMMAR RULE FOR CALTUR ON A DEMO_GRID.
```

```
STRAN INTERPRETER ON MAY 24,1969 AT 14:48:54.090 PAGE  9

        %CALCOM;4 IS THE GRAMMAR RULE FOR CALTUR ON A DEMO_GRID.
        (FUNCTION,GRAMMAR RULE FOR&ON)
        (NOUN ROOT,GRAMMAR RULE FOR&ON)
        (GRAMMAR RULE FOR&ON,%CALCOM;4,CALTUR,DEMO_GRID)
INPUT...%CALCON;3 IS THE GRAMMAR RULE FOR CONTUR ON A DEMO_GRID.

        %CALCON;3 IS THE GRAMMAR RULE FOR CONTUR ON A DEMO_GRID.
        (GRAMMAR RULE FOR&ON,%CALCON;3,CONTUR,DEMO_GRID)
INPUT...(GRAMMAR)
        FOLLOWING INPUT STORED AS GRAMMAR RULES.
INPUT...(%AL2:3=<,4,4,0,0,1,,50.0,1,4>)
INPUT...(%CALCOM=%CALCON+%MOVE)(%AL2:4=<,4,4>))
        END OF GRAMMAR.

INPUT...GRAPHICAL OUTPUT GOES ON THE CALCOMP PLOTTER.

        GRAPHICAL OUTPUT GOES ON THE CALCOMP PLOTTER.
        (UNIQUE,CALCOMP PLOTTER)
        (VERB ROOT,GOES ON)
        (GOES ON,GRAPHICAL OUTPUT,CALCOMP PLOTTER)
INPUT...M6350 IS THE PROBLEM_NUMBER.  2024 IS THE PROGRAMMER_NUMBER.

        M6350 IS THE PROBLEM_NUMBER.
        (FUNCTION,PROBLEM_NUMBER)
        (PROBLEM_NUMBER,M6350)

        2024 IS THE PROGRAMMER_NUMBER.
        (FUNCTION,PROGRAMMER_NUMBER)
        (PROGRAMMER_NUMBER,2024)
INPUT...PSI IS A DEMO_GRID.

        PSI IS A DEMO_GRID.
        (DEMO_GRID,PSI)
```

```
STRAN INTERPRETER ON MAY 24,1969 AT 14:48:56.890 PAGE 10


INPUT...))))

        FOLLOWING ARE DEDUCTIONS FROM DECLARATIONS.
         (GOES ON,CONTOUR MAP,CALCOMP PLOTTER)
         (GOES ON,GRAPH,CALCOMP PLOTTER)
         (GOES ON,GEOGRAPHIC OUTLINE MAP,CALCOMP PLOTTER)
         (REAL,PSI)
         (ARRAY,PSI)

        FOLLOWING INITIALIZATION STATEMENTS GENERATED.

        FOLLOWING IS EXPANSION OF COMMAND...OPEN THE CALCOMP PLOTTER
        CALL NEWPLT('M6350','2024','VELLUM','BLACK')
        REAL PSI(4,4)

        FOLLOWING INPUT IMPERATIVE STATEMENTS PROCESSED.
INPUT...GET DATA FOR PSI; PRODUCE CONTOURS FOR PSI; END;

        FOLLOWING IS EXPANSION OF COMMAND...GET DATA FOR PSI
        CALL READ(5,'(24F3.0/23F3.0)',PSI,1,IDIM*JDIM)

        FOLLOWING IS EXPANSION OF COMMAND... PRODUCE CONTOURS FOR PSI
        CALL CALTUR(PSI,4,4)

CONTOUR PROGRAM HAS BEEN ENTERED.
MINIMUM IS  0.0        . MAXIMUM IS  2.0000E 02. CONTOUR INTERVAL IS  2.0E 01.
CONTOUR PROGRAM TOOK    0.63 SECONDS.
        CALL PLOT1(14.0,0,-3)
        END
        CALL ENDPLT
```

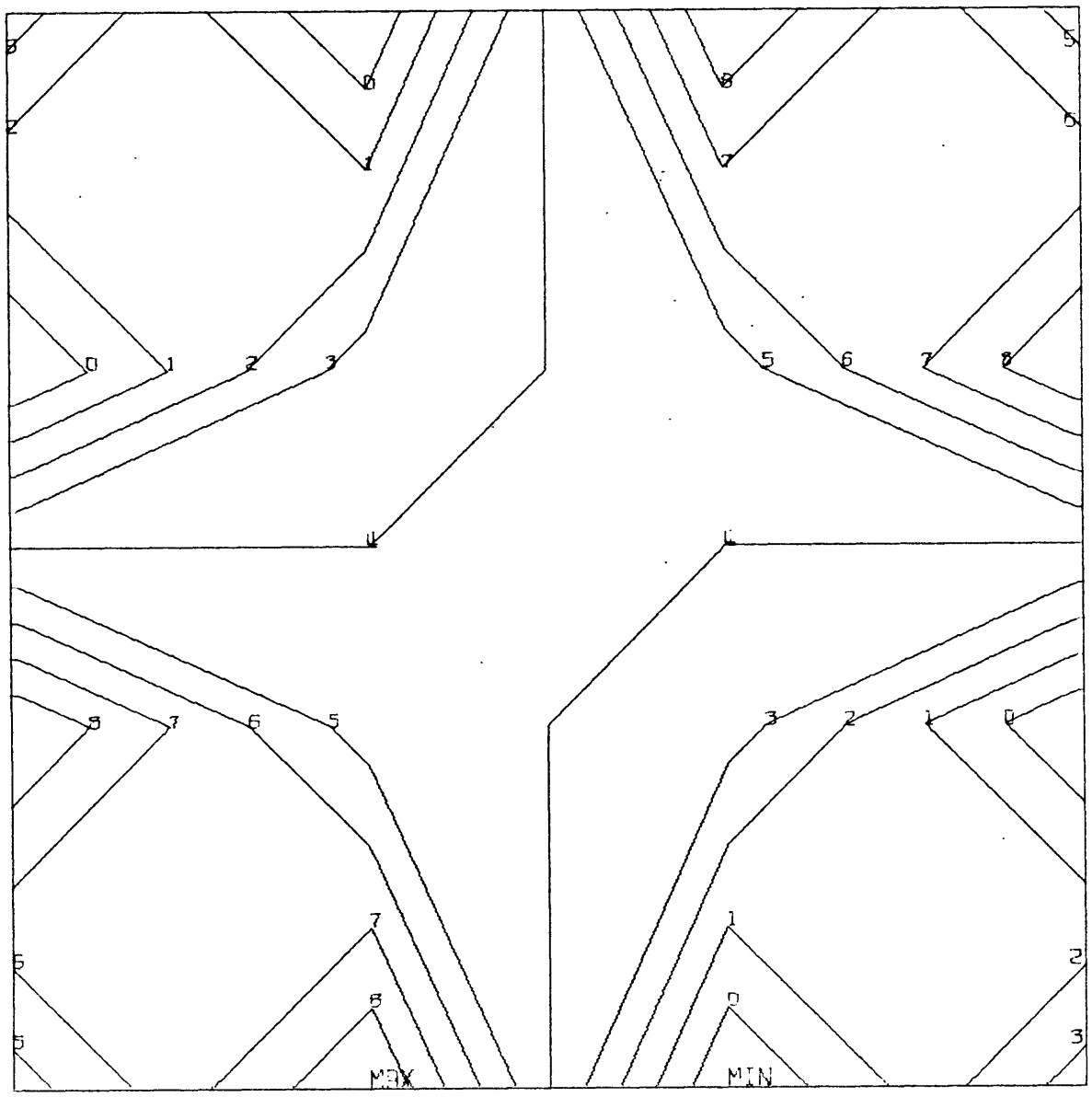


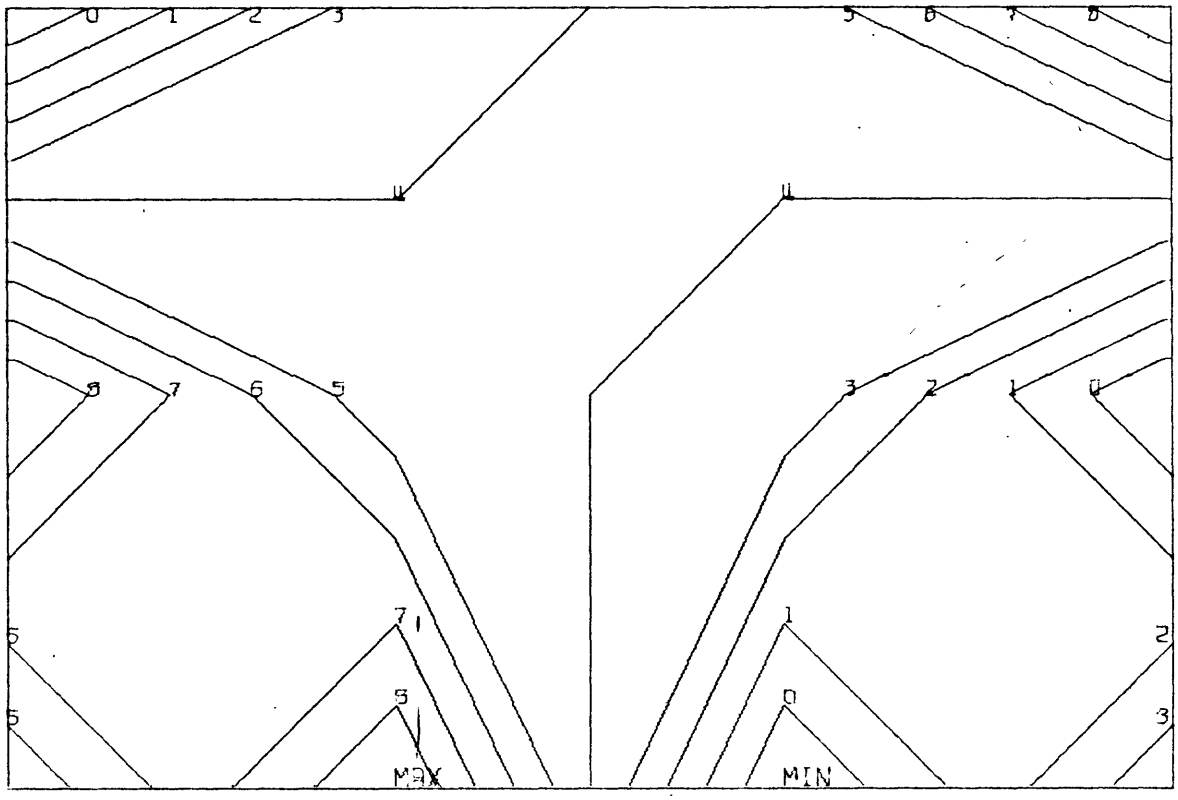
Fig. 8.3 Calcomp plotter contour map of a DEMO_GRID, produced by Application Example 4.

The final example shows how a data object can be handled when, unlike the previous four examples, data of its exact type is unlikely to occur again in the future. In such a case it appears cumbersome to create a name for a type of grid and to make a number of declarations about grids of that type. Thus, this example shows how a data object whose type is unspecified may be handled by the same commands that we have been using to work on data objects with a prespecified type. Since no data type is specified, this example shows the default operation of the commands. It seems clear that in such a default case the command processor may not provide much simplification of the user's task, and an experienced user might decide to code the call statements himself, instead of letting the command processor do it for him.

The declarations of this example assert that PSI is an array and that its dimensions are four by three. It is not necessary to declare that PSI is real, for that is the default assumption for any array under this COMS.

```
STRAN INTERPRETER ON MAY 24,1969 AT 15:01:41.000 PAGE  8

BEGIN INTERPRETING RULES.

        FOLLOWING INPUT DECLARATIONS PROCESSED.

INPUT...GRAPHICAL OUTPUT GOES ON THE CALCOMP PLOTTER.

        GRAPHICAL OUTPUT GOES ON THE CALCOMP PLOTTER.
        (UNIQUE,CALCOMP PLOTTER)
        (VERB ROOT,GOES ON)
        (GOES ON,GRAPHICAL OUTPUT,CALCOMP PLOTTER)
INPUT...M6350 IS THE PROBLEM_NUMBER.  2024 IS THE PROGRAMMER_NUMBER.

        M6350 IS THE PROBLEM_NUMBER.
        (FUNCTION,PROBLEM_NUMBER)
        (PROBLEM_NUMBER,M6350)

        2024 IS THE PROGRAMMER_NUMBER.
        (FUNCTION,PROGRAMMER_NUMBER)
        (PROGRAMMER_NUMBER,2024)
INPUT...PSI IS AN ARRAY.   4 IS THE FIRST DIMENSION OF PSI.

        PSI IS AN ARRAY.
        (NOUN,ARRAY)
        (ARRAY,PSI)

         4 IS THE FIRST DIMENSION OF PSI.
        (NOUN ROOT,FIRST DIMENSION OF)
        (FIRST DIMENSION OF,4,PSI)
INPUT...3 IS THE SECOND DIMENSION OF PSI.        ))))

        3 IS THE SECOND DIMENSION OF PSI.
        (FUNCTION,SECOND DIMENSION OF)
```

```
STRAN INTERPRETER ON MAY 24,1969 AT 15:01:43.160 PAGE  9

          (NOUN ROOT,SECOND DIMENSION OF)
          (SECOND DIMENSION OF,3,PSI)

          FOLLOWING ARE DEDUCTIONS FROM DECLARATIONS.
           (GOES ON,CONTOUR MAP,CALCOMP PLOTTER)
           (GOES ON,GRAPH,CALCOMP PLOTTER)
           (GOES ON,GEOGRAPHIC OUTLINE MAP,CALCOMP PLOTTER)

          FOLLOWING INITIALIZATION STATEMENTS GENERATED.

          FOLLOWING IS EXPANSION OF COMMAND...OPEN THE CALCOMP PLOTTER
          CALL NEWPLT('M6350','2024','VELLUM','BLACK')
          REAL PSI(4,3)

          FOLLOWING INPUT IMPERATIVE STATEMENTS PROCESSED.
INPUT...GET DATA FOR PSI; PRODUCE CONTOURS FOR PSI; END;

          FOLLOWING IS EXPANSION OF COMMAND...GET DATA FOR PSI
          CALL READ(5,'(24F3.0/23F3.0)',PSI,1,IDIM*JDIM)

          FOLLOWING IS EXPANSION OF COMMAND... PRODUCE CONTOURS FOR PSI
          CALL CALTUR(PSI,IDIM,JDIM)

CONTOUR PROGRAM HAS BEEN ENTERED.
MINIMUM IS  0.0        . MAXIMUM IS  2.0000E 02. CONTOUR INTERVAL IS  2.0E 01.
CONTOUR PROGRAM TOOK   0.50 SECONDS.
          CALL PLOT1(14.0,0,-3)
          END
          CALL ENDPLT
```



MAXIMUM VALUE IS 200.0
MINIMUM VALUE IS .0
CONTOUR INTERVAL IS 20.0

Fig. 8.4 Calcomp plotter contour map of a special 4x3 grid, produced by Application Example 5.

CHAPTER IX

CONCLUSIONS

Throughout the preceding material, we have described a number of ways that the system foundation elements presented can be used, alone and together. It has been shown that each element provides the necessary tools for dealing with a particular phase of the problem environment. Also the examples given have demonstrated how more and more sophisticated COMS implementations may be produced by sophisticated users and systems programmers. This attribute of extendability is one of the most important aspects of the design of COMS. It also has been shown how application program libraries can be utilized by COMS, and that COMS makes the members of the library much more accessible to the user. Furthermore, it has been seen that COMS is no solution to bad programming in an application program, but that the combination of well written library programs and the accessibility available through COMS can provide great problem solving power.

We have described the advantages of COMS in the foregoing material. At this point some attention should be paid to its liabilities. A criticism that can be presented against the COMS system and the foundation elements which comprise it, is inefficiency. The COMS system is not frugal in its use of execution time and core storage space (this criticism does not apply to the library programs presented, which are quite efficient). The reasons for this inefficiency are implicit in the whole design philosophy of the COMS system. The

coding of the foundation elements was done in PL/1 in order to achieve easy development and modification through compactness and clarity of the code. An interpretive mode of operation (without any preprocessing of rules, so that rules and data could be handled identically) was chosen in order to allow quick and easy modification (by others) of the system at any time desired, and to allow different versions of the system to be available simultaneously. The design of the associative memory mechanism as a completely general purpose storage for n-tuples of character strings also contributed to inefficiency. Thus, in most cases where a choice had to be made between efficiency and flexibility, efficiency lost. This should not alarm the reader into thinking that the COMS system is somewhat of a "lumbering ox", for most of the examples presented were accomplished in a few seconds of computer time, and at no time has the author had any trouble with core storage (400K bytes) becoming overloaded. However, if this system were to be used by an ordinary community of users in a smaller computer, such problems would become more manifest. Thus, when such use becomes imminent, a certain amount of tuning and optimization will be in order.

Optimization can be done in a number of ways. One of the simplest means of optimization will be to allocate as little storage for rules and n-tuples as is absolutely necessary for a particular implementation. Because of the flexibility of the interpreter, this is easily done at the beginning of any run when such data collections are to be created. Another means of optimization will be to decrease the number of forms under which an n-tuple is stored, and to compact the storage of char-

acter strings in the dictionaries. The primary means by which speed may be increased, is to recode certain heavily used subprograms (particularly those carrying out hash-coding) in assembly language. In any such recoding, care should be exercised to produce an assembly language subroutine which appears externally identical to its PL/1 coded brother. Such care will allow any later ideas about changes to the system to be quickly implemented and tested in PL/1 and, when proved useful, to be transferred to the assembly language version subsequently.

Another form of optimization and improvement which might need to be undertaken if COMS were to be regularly used, would be programs to allow both dynamic allocation and garbage collection of core storage space. At present dynamic allocation of space is carried out, but no facilities have yet been provided for clearing away application programs or numeric arrays which are no longer needed. Such facilities would not contribute to the conceptual framework of the system, but would contribute to its useability.

Future work might also include an investigation of the usefulness of simultaneously using more than one distinct collection of n-tuples (model) for the associative memory. An example of a useful pair of models would be the ordinary COMS self-model and a model of the users special problem environment, including a description of his level of ability and need for help and advice. The users model might also contain sentences to override normal system conventions, stated in

the COMS model, which the user does not like. To extend that concept, one might even be able to define a whole hierarchy of models, each able to override the statements of those below, with the COMS model at the bottom serving as the basic set of assertions. Extension of the modeling capabilities of COMS in such ways appear to hold the most exciting possibilities for future development.

The design philosophy of COMS has been such as to emphasize flexibility, modularity, simplicity and clarity over efficiency. This has been done because computer software design resembles artistic creation in many ways. Both require the exercise of technical knowledge, craftsmanship, skill and patience, but most important both involve an element of serendipity. Thus, if the artist or software designer works in an unforgiving medium (eg. assembly language or granite), then a brilliant inspiration occurring midway through his labors may be nearly impossible to carry out. However, if he works in a more forgiving medium (eg. PL/1 or clay), and works in a lucid and organized manner, he may discover that the process of his labor teaches and inspires him to the extent that the final product bears little resemblance to his original concept. Such change need not be feared, and will not produce a monstrosity, if the conceptual foundation of the work is solid.

APPENDIX A

FORTRAN PROGRAM ORGANIZATION FOR EFFECTIVE CONTROL

A.1 Introduction

In this appendix we will examine some methods by which a FORTRAN program may collect directive information from outside itself. In each of these methods the amount of information collected may vary from minimal (that information absolutely essential for correct execution) to maximal (settings for all optional parameters). In the case where less than maximal information is collected, default assumptions must be used to produce values for unspecified optional parameters. A FORTRAN program which uses these methods will be much more flexible and easier to use than it would have been without them. The beginning user of the program will find much less mandatory information required and the advanced user will find he has the ability to exert much more control over the program. The methods to be described utilize NAMELIST input (to be explained below) and variable length argument lists to accomplish these goals.

The first method, named SET-RESET, uses NAMELIST input to direct a program which must make a number of passes through itself, each pass requiring a different setting of the control parameters. An example of such a program would be a numerical model in the fluid sciences, with each pass being a forward time integration of specified duration under specified conditions.

The second method of information collection is the use of the variable length argument list. This method is primarily of use in utility subroutines (widely used subroutines for carrying out basic tasks). The third method is an extension of the second, allowing the length of the set of optional arguments to be almost indefinitely extended through the use of NAMELIST input. An example of a program using these methods to advantage is the contouring program for the Calcomp plotter, written by the author. An inexperienced user of this program can produce a contour map by giving the minimum three arguments (eg. CALL CALTUR(PSI,10,10)) while an experienced user may provide values for any of over two hundred parameters.

A.2 SET-RESET: A Method of Program Organization Using NAMELIST Input.

In recent years a new kind of input-output statement has been introduced in many algebraic compiler languages. This type of I-O is variously known as simplified, data-directed or namelist input-output, depending upon whether the language is MAD, PL/I or FORTRAN (Only a few FORTRAN compilers include this option. It is <u>not</u> part of USA FORTRAN). Since the term namelist is the most distinctive of the three, we will refer to all three as NAMELIST I-O.

The NAMELIST I-O method makes use of an internal symbol dictionary to allow values to be assigned to or dumped from specified variables and arrays at execution time. NAMELIST I-O is extremely

useful, and a programmer who becomes facile in its use finds himself loath to utilize languages which do not contain it.

Like most other programming statements, namelist I-O can be used most effectively under certain modes of program organization, and for certain kinds of specialized tasks. When used as a straight replacement for a formatted I-O statement it tends to be inefficient, and to gain the programmer only the elimination of the format statement.

Many astute programmers appear to have stumbled upon methods of program organization which make namelist I-O an effective tool. However, the description of these methods appear to have been carried only by word of mouth, and many in the field have not been fortunate enough to hear about them. Sadly, this latter group contains many of the people who write the manuals and primers purporting to teach programmers how to use MAD, PL/I and FORTRAN. The result is that almost any manual or primer for these languages which one is likely to pick up, will do a very bad job of explaining the use of this type of I-O. The few examples given usually only point out the lack of a format statement and demonstrate how the data cards are punched. Rarely is it pointed out that since only a subset of the variables in the symbol dictionary need be read in, this type of I-O is very useful for modifying default values of a large set of independent variables and control parameters (A notable exception is McCracken, Daniel D., FORTRAN with Engineering Applications, pp. 174-5). Even

rarer is a mention of the fact that successive passes through the namelist read statement and associated program can allow specified modifications to be made to preceding settings, resulting in a cumulative specification of values to produce a step by step march through a problem space. Nor is it ever emphasized that NAMELIST I-O can produce a program which an inexperienced user can use to produce default results immediately without having to create any input values, while the experienced user of the program has access to all of the internal control parameters, including those that no one would dream of making accessible through formatted I-O because of the cost and inconvenience in normal use.

Because this writer feels that NAMELIST I-O has extreme importance in many applications, because it has been under-utilized by programmers with whom the writer is acquainted, and finally because the writer had to stumble upon this approach completely without outside guidance, the following material is presented in hopes that other programmers will not find it necessary to do the same stumbling and that a certain amount of light can be shed on this under-examined area of programming. Perhaps also some discussion can be generated among programmers about new methods of application of NAMELIST I-O.

The method of program organization developed for utilizing NAMELIST I-O has been named the SET-RESET method, in order to have something to refer to. The attaching of a name to the method is not intended to create an aura of something newly discovered (although

such was certainly the case for the writer), but to provide clarity in the discussion that follows. The following is a simple system 360 FORTRAN IV G program which illustrates the most basic form of the SET-RESET method of organization and its use of NAMELIST.

SOURCE CARDS:

```
      NAMELIST/INPUT/RESET,I,J
      LOGICAL RESET
C THE FOLLOWING ARE THE DEFAULT VALUES OF THE NAMELIST
C VARIABLES WHICH ARE SET AT THIS TIME.
  1   I = 1
      J = 1
      RESET = .FALSE.
C END OF DEFAULT VALUES.
  2   IF (RESET) GO TO 1
C READ NEXT 'INPUT' STRING.
      READ (5,INPUT,END=3)
C WRITE OUT THE SETTINGS OF ALL NAMELIST VARIABLES FOR THIS PASS.
      WRITE (6,INPUT)
C ANY CODE UTILIZING I AND J IS INSERTED HERE.
C A POSSIBLE EXAMPLE IS THE FOLLOWING.
      NAMELIST/OUTPUT/K
      K = I*J
      WRITE (6,OUTPUT)
C END OF INSERTED CODE.
      GO TO 2
  3   CALL EXIT
      END
```

DATA CARDS:

```
      &INPUT I=5, J=5782 &END
      &INPUT I=2400, RESET=T &END
      &INPUT &END
```

The data cards shown will cause three passes to be made through the program before EXIT is called. The following are the settings of I,J and RESET for each pass.

PASS	VALUE OF I	VALUE OF J	VALUE OF RESET
1	5	5782	.FALSE.
2	2400	5782	.TRUE.
3	1	1	.FALSE.

FLOW DIAGRAM:

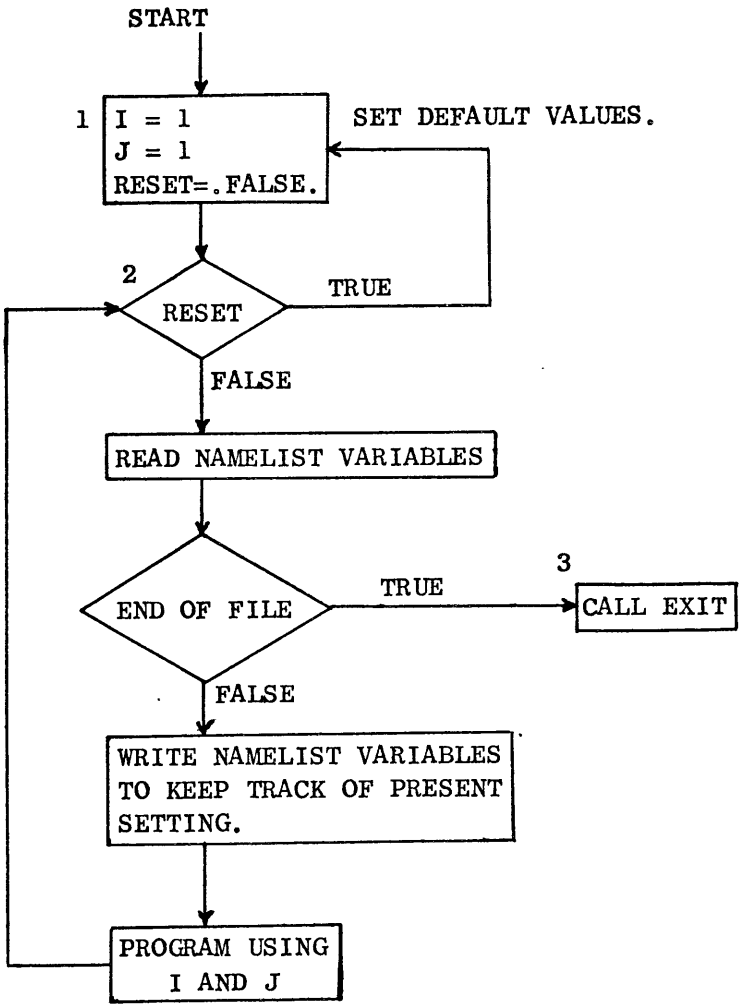


Fig. A.1 Flow diagram for first SET-RESET example.

The points to be noticed in this simple example are:

(1) The set of independent variables and control parameters are program initialized.

(2) Any subset of them may be modified by input data.

(3) Inclusion of a logical control parameter (RESET) among the other variables makes it possible to reset all to their initial settings at the beginning of any pass through the program.

(4) If the variables have not been reset, their initial values for any pass will be those left from the previous pass. This allows inputs to be cumulative, each pass serving as one step of a march through the problem space.

The fact that any subset (including the empty set and the whole set) of variables may be modified provides the extreme flexibility of NAMELIST which makes the SET-RESET method possible. The method derives further usefulness from the fact that a researcher usually investigates a problem space in incremental steps, making small changes in a few independent variables, and only rarely jumps to an entirely new region by making larger changes in a large set of independent variables.

In the example, the SET-RESET method of organization and its use of NAMELIST may not appear particularly advantageous. However, when larger sets of independent variables and control parameters are involved, the program which uses them can be controlled in a much more flexible manner. This increase in flexibility occurs because the set of subsets of n parameters, which may be specified, numbers 2^n.

A more complex and useful example of the SET-RESET method is the following.

SOURCE CARDS:

```
      COMMON/DATA/IDIM,JDIM,MODE,PHI(50,50)
1     CALL INIT
C  CODE TO UTILIZE DATA VARIABLES IS INSERTED HERE.
C  EXAMPLE OF POSSIBLE USE OF MODE IS:  GO TO (2,3,4,5), MODE
      GO TO 1
      END
      SUBROUTINE INIT
      LOGICAL RESET,PUPHI,RDPHI,RDFORM,RDPRT,FIRSTM
      COMMON/DATA/IDIM,JDIM,MODE,PHI(50,50)
      NAMELIST/INPUT/RESET,PUPHI,RDPHI,RDFORM,RDPRT,NTAPE,MODE,IDIM,JDIM,
     1PHIA,PHIB,PCONST,PHASEI,PHASEJ,WAVNOI,WAVNOJ
      NAMELIST/PERTRB/PHI
      DIMENSION FORMAT(20),FMT(2)
      DATA FMT,FIRSTM/4H(10F,4H8.2),.TRUE./
      IF (FIRSTM) GO TO 1
      IF (PUPHI) WRITE (7,FORMAT) ((PHI(I,J),I.1,IDIM),J=1,JDIM)
      IF (.NOT.RESET) GO TO 2
1     FIRSTM = .FALSE.
      RESET = .FALSE.
      PUPHI = .FALSE.
      RDPHI = .FALSE.
      RDFORM = .FALSE.
      RDPRT = .FALSE.
      NTAPE = 5
      MODE = 1
      IDIM = 20
      JDIM = 10
      PHIA = 0.0
      PHIB = 0.0
      PCONST = 1.0
      PHASEI = 0.0
      PHASEJ = 0.0
      WAVNOI = 2.0
      WAVNOJ = 1.0
      FORMAT(1) = FMT(1)
      FORMAT(2) = FMT(2)
2     READ (5,INPUT,END=6)
      WRITE (6,INPUT)
      IF (RDPHI) GO TO 4
      FISIZ = IDIM-1
      FJSIZ = JDIM-1
      DO 3 J=1,JDIM
      FJVAL = J-1
      FACJ = SIN(FJVAL*WAVNOJ*3.14159/FJSIZ + PHASEJ)
      DO 3 I=1,IDIM
```

```
3       PHI(I,J)=PHIA*FJVAL+PHIB + PCONST*FACJ*
       1   SIN(FLOAT(I-1)*WAVNOI*3.14159/FISIZ+PHASEI)
        GO TO 5
4       IF (RDFORM) READ (5,7) FORMAT
        READ  (NTAPE,FORMAT) ((PHI(I,J),I=1,IDIM),J=1,JDIM)
5       IF (RDPRT) READ (5,PERTRB,END=6)
        RETURN
6       CALL EXIT
7       FORMAT(20A4)
        END
```

DATA CARDS:

```
&INPUT PUPHI=T,RDPRT=T, &END
&PERTRB PHI(15,3)=7.4, PHI(2,7)= -20.1 &END
&INPUT IDIM=2,JDIM=2,RDPHI=T,RDFORM=T &END
(20F4.2)
7.2 6.2 3.5 8.7
&PERTRB PHI(1,1)=10.5 &END
```

FLOW DIAGRAM OF INIT SUBROUTINE:

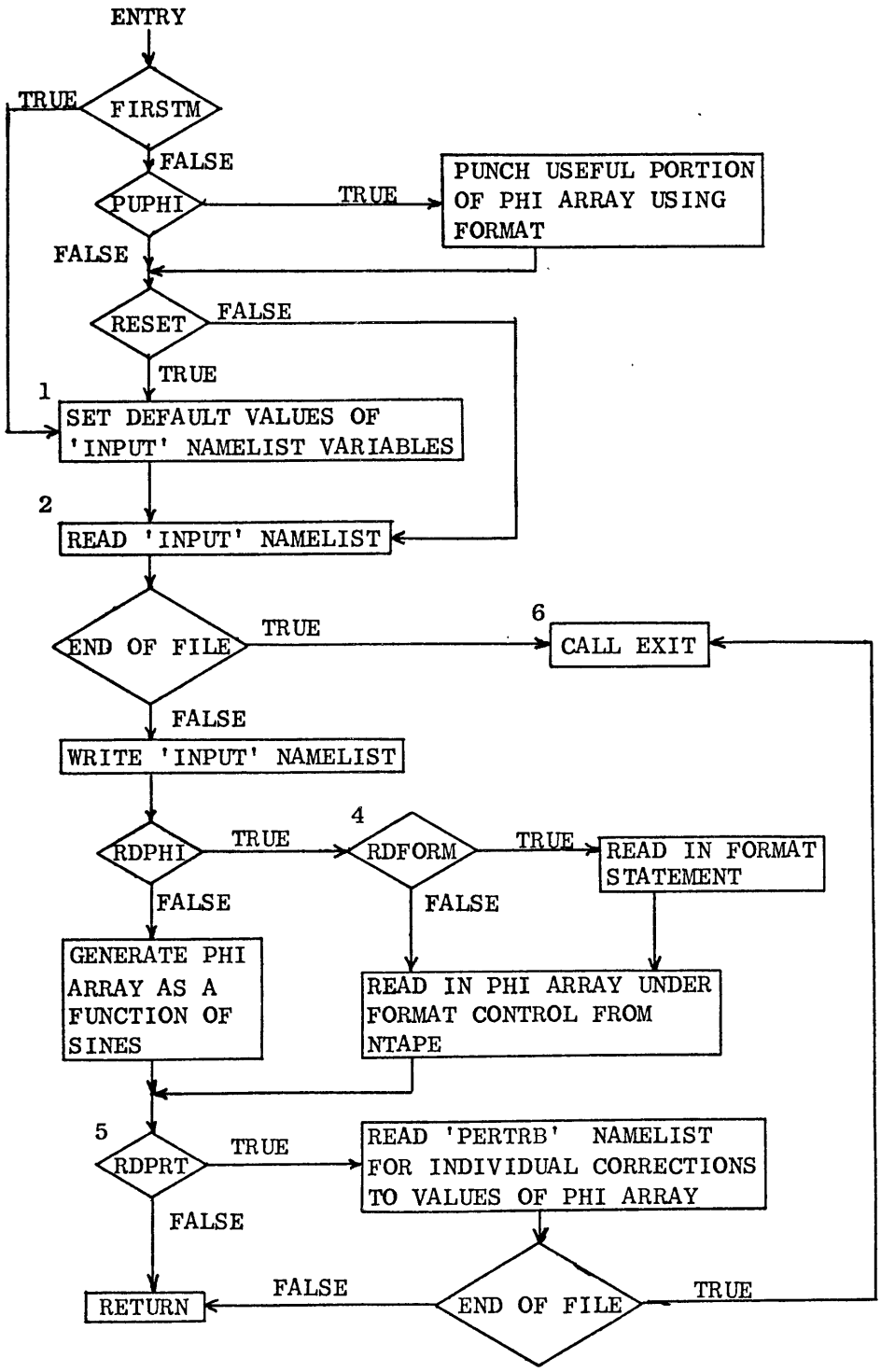


Fig. A.2 Flow diagram for second SET-RESET example.

Many new refinements have been introduced in this second example. Some of the more important ones are the following.

(1) The SET-RESET method has been localized in the subroutine INIT, with independent variables passed to the user program through a named common block.

(2) The array PHI may be initialized either by computing it as a function or by reading in formatted data under an arbitrary format from a specifiable device.

(3) A second NAMELIST has been added to allow introduction of individual values into the interior of the array PHI after it is initialized.

As it stands, with no user code entered in the main program, the example serves as a simple program for generating, editing and correcting large formatted data decks.

The SET-RESET method, as can be seen, encourages the user to write programs which are extremely data directed. Even at the level of this example, without any useful program to direct, the method produces a program which acts very much like an interpreter. The control information must be passed as FORTRAN data types, which prevents the input from appearing truly interpretable, but the flexibility of control available is nearly the same.

The SET-RESET form of organization has proved especially valuable as a control method for large numerical models in fluid dynamics.

It seems clear that this method has much wider applicability, however, in any program which has many modes of operation, or some uncertainty as to the exact best setting of various independent parameters. The SET-RESET method allows the programmer to give the parameter a 'good guess' default value until experimentation has produced a best value, at which time a single recompilation sets the default value correctly. Possible refinements to the basic SET-RESET method seem nearly endless. It is a particularly fruitful form of organization when a program is being used in an on-line man-machine interaction environment such as CTSS* at M.I.T. One very successful program written in MAD for use under CTSS uses the SET-RESET method to collect its control parameters. This program displays an orthographic projection of the globe on the face of a storage CRT. Inexperienced users of the program are able to accomplish the plotting of a map of North America by simply typing an asterisk, indicating an empty argument string. More experienced users can modify the values of any of a large number of control parameters to plot the globe in other projections and different magnifications and orientations, occasionally producing results which were unforeseen at the time the program was written. The SET-RESET method can successfully bridge the gap between interactive and batch processed use, and is not awkward in either environment.

---

*The Compatible Time Sharing System - the IBM 7094 time sharing system at M.I.T.

A.3 Subroutine Organization for Variable Length Argument Lists.

The use of variable length argument lists by subroutines in the library is not directly related to COMS, except that versatile, easy to use and flexible application programs in its library will make COMS a more useful system. An application program that has sixteen mandatory arguments cannot be considered flexible or easy to use, for even experienced users will have trouble remembering what all those arguments are and in what order they appear. Thus, the material of this section is presented to show the methods which should be used by application program authors in order to write easily used programs. It is not necessary to use these methods for a program to be put in the COMS library, but they will produce a better program. The easiest way to present the material is by giving an example, and then explaining it.

```
      SUBROUTINE EXAMPL(ARRAY,/ISIZE/,/JSIZE/,/OPTRG1/,/OPTRG2/,...)
      DIMENSION ARRAY(ISIZE,JSIZE)
      DATA ARG2,ARG3/0.0,5.0/
      ARG1 = 10.0
       .
       .
       .
      CALL NUMP(NARG)
      IF(NARG.GT.2) TO TO 101
      WRITE(6,100)
  100 FORMAT('TOO FEW ARGUMENTS PASSED TO EXAMPL.')
      RETURN
```

```
101   NARG = NARG - 2
      GO TO(1,2,3,...),NARG
       .
       .
       .
  4   IF(OPTRG3.EQ.0) GO TO 3
      ARG3 = OPTRG
  3   ARG2 = OPTRG2
  2   ARG1 = OPTRG1
  1   CONTINUE
       .
       .
       .
      body of program
      END
```

The subprogram NUMP is a simple assembly language routine which counts the number of arguments passed to the program it is called by, and places that integer number in NARG. If the number of arguments is less than three, the number of arguments which the programmer has decided are mandatory, an error message is written and control is returned. If enough arguments have been given, a computed go-to is executed and exactly the number of optional arguments passed are collected. An especially important point to note is that the values of all optional arguments are initialized to default values. Two methods are used. ARG1 will be initialized to its default value of 10.0 (by assignment statement) every time the program is entered, and modified if any value for it is specified in the argument list. ARG2 is initialized when the program is loaded (by data statement), modified if any value is specified for it in an argument list, and

that modified value will serve as the default value for subsequent calls to the program. ARG3 is initialized at loading time, but only nonzero arguments will cause its modification.

As can be seen, the methods can produce a program which is easy to use for a new user, but also versatile and flexible in the hands of a more knowledgeable user.

A.4 Subroutine Organization for Optional Argument Transmission via Namelist.

In Section A.2 SET-RESET is described. That method is designed to use namelist input for control of a program which makes multiple passes through a numerical model, or other mechanism for testing a particular problem space. Namelist input proves useful for making adjustments in a few independent variables, from a large set, at the beginning of each pass. If not set to a new value, a variable continues with the same value, left from the previous pass. For the initial pass that value is the default value which has been program initialized.

Success achieved using namelist input to control numerical models suggested that a similar facility might prove just as useful operating between calling and called programs, to specify values for variables which need to be modified very infrequently. Under such an approach the calling program must provide a sequence of pairs of variable names and the values to be assigned to those variables.

An example might be:

& VARB1 = 5.0, ALPHA = 81.5, TOTAL = 506.7 &

An advantage of this form of argument specification over the argument list method, described in the preceding section, is that no order is implied upon the variables. In an argument list, if the fifth variable is the only one which really needs to be specified, values must be given for the preceding four in order to indicate that it is the fifth variable which is being set. Also in an argument list, no great number of arguments can be allowed without confusion arising in the mind of the user as to just how far down the argument list he has progressed when he must provide the next value to go in the list. No such confusion can arise with a namelist specification of arguments, for each is clearly marked with the name of the variable to which the value is to be assigned. Furthermore, because having very large sets of variables to which values can be assigned will not create any confusion in the mind of the user, it is possible to give the user the ability to modify the values of all of the control variables of a complex subprogram. This can give a user much greater control over a program than he has ever experienced before.

Disadvantages of the namelist method of argument transmission are primarily in its inefficiency relative to argument lists. Manipulation of character strings and dictionary look-up of variable names are much slower processes than the retrieval of an argument from a list. Thus, for mandatory arguments and frequently specified

optional arguments, the variable length argument list method of the previous section is superior. Further down the argument list, where variables are much less frequently specified, the flexibility (at a cost) of namelist argument specification begins to be more appropriate.

The following program is an extension of that given in the preceding section, showing how namelist argument specification may be incorporated without slowing the speed of execution of the library program under normal circumstances.

```
      SUBROUTINE EXAMPL(ARRAY,/ISIZE/,/JSIZE/,/OPTRG1/,/OPTRG2/,...)
      DIMENSION ARRAY(ISIZE,JSIZE)
      NAMELIST/INPUT/NTAPE,ARG4,ARG5,ARG6
      DATA ARG2, ARG3, ARG4, ARG5, ARG6/0.0,5.0,3.148,7.69,10.1/
      DATA NTAPE/6/
      ARG1 = 10.0
       :
      CALL NUMP(NARG)
      IF(NARG.GT.2) GO TO 101
      WRITE(NTAPE,100)
100   FORMAT('TOO FEW ARGUMENTS PASSED TO EXAMPL.')
      RETURN
```

```
101   NARG = NARG - 2
      GO TO(1,2,3,...),NARG
       .
       .
       .
  4   IF(OPTRG3.EQ.0.0) GO TO 3
      ARG3 = OPTRG3
  3   ARG2 = OPTRG2
  2   ARG1 = ABS(OPTRG1)
  1   IF(OPTRG1.LT.0.0) READ (5,INPUT)
       .
       .
       .
      body of program
      END
```

This program works almost the same as the example of Section A.3, except that when the fourth argument of its calling sequence is negative, a namelist read occurs, and values can be read into the normally unmodified variables NTAPE, ARG4, etc. This means that a user now has access to variables to which access could never before be allowed.

APPENDIX B

EXAMPLES OF STRAN LANGUAGE APPLICATIONS

These examples are presented here, instead of within the main body of the thesis, because they are not examples of the utilization of a program library. That is, they are not examples of particular implementations of COMS. These examples illustrate the kind of operations which can be carried out using the STRAN language. Many of these operations have been used to advantage in the sophisticated version of COMS which is presented at the end of the thesis. These examples should also suggest other applications and systems to which the foundation elements may usefully be applied. The reader should note the brevity and simplicity of the rule set in relation to the complexity of the task in each case.

B.1 Example 1

On one occasion when two rule sets were to be used together as subprocedures in the solution of a large problem, it was discovered that the name PRINT had inadvertantly been used as a rule name in one set and a variable name in the other. This meant that the rule contents would quickly become obliterated on its first use as a variable. The solution found was to write a short rule set to change every occurrence of the name PRINT in the rules where it was used as a variable, to the name OUT. Thus, the data cards in this case were a rule set, being operated on by another rule set. The trans-

formed results were punched as a result of a (PUNCH) command, which tells the STRAN processor to punch all output requested by the rules.

This example illustrates the ease with which context editing programs can be written in STRAN. Only the rule CONV1 is doing any transformation of the character string. The other two rules simply handle input and output. Readers familiar with the context editing facilities in CTSS will note the similarity in effect to the "change" command which is available there. It appears that the STRAN language could serve as a very flexible basis for the implementation of experimental on-line context editors in the future.

BEGIN READING RULES.

```
INPUT...(CONVERT(*INPUT|$|)CONV1)
INPUT...(CONV1(INPUT|$+'PRINT'+$|=INPUT|1+'OUT'+3|)CONV2,CONV1)
INPUT...(CONV2(INPUT|$|=*INPUT|1|*INPUT|' '|)CONVERT)
INPUT...(PUNCH)
INPUT...(CONVERT)
```

```
INPUT...(UN2(=&S|'SUBSET OF'+2+1|*PRINT|' (SUBSET OF,'+2+','+1+')'|)UN3)
        (UN2(=&S|'SUBSET OF'+2+1|*OUT|' (SUBSET OF,'+2+','+1+')'|)UN3)

INPUT...(UN3(=&S|'SUBSET OF'+3+1|*PRINT|' (SUBSET OF,'+3+','+1+')'|)UN1)
        (UN3(=&S|'SUBSET OF'+3+1|*OUT|' (SUBSET OF,'+3+','+1+')'|)UN1)

INPUT...(LEHALF(&F1|'LEFT HALF OF'+$+$|=*PRINT|'EXECUTING LEFT HALF.'|)END,LEH1)
        (LEHALF(&F1|'LEFT HALF OF'+$+$|=*OUT|'EXECUTING LEFT HALF.'|)END,LEH1)

INPUT...(LEH1(&A1|1+2|)END,LEH2)
        (LEH1(&A1|1+2|)END,LEH2)

INPUT...(LEH2(&F2|2+$+$|)LEH1,LEH3)
        (LEH2(&F2|2+$+$|)LEH1,LEH3)

INPUT...(LEH3(&A2|3+4|=&S|1+3|*PRINT|' ('+1+','+3+')'|)LEH1,LEH3)
        (LEH3(&A2|3+4|=&S|1+3|*OUT|' ('+1+','+3+')'|)LEH1,LEH3)

INPUT...(RIHALF(&F1|'RIGHT HALF OF'+$+$|=*PRINT|'EXECUTING RIGHT HALF.'|)END,RIH1)
        (RIHALF(&F1|'RIGHT HALF OF'+$+$|=*OUT|'EXECUTING RIGHT HALF.'|)END,RIH1)

INPUT...(RIH1(&A1|1+2|)END,RIH2)
        (RIH1(&A1|1+2|)END,RIH2)

INPUT...(RIH2(&F2|2+$+$|)RIH1,RIH3)
        (RIH2(&F2|2+$+$|)RIH1,RIH3)

INPUT...(RIH3(&A2|3+4|=&S|1+4|*PRINT|' ('+1+','+4+')'|)RIH1,RIH3)
        (RIH3(&A2|3+4|=&S|1+4|*OUT|' ('+1+','+4+')'|)RIH1,RIH3)
```

B.2 Example 2

When the arithmetic equation evaluator (which is written as a self contained external subroutine in PL/I) was being tested, a special rule set was created to accept equations from cards and print out the result of the evaluation. Evaluations are carried out in both fixed and floating point, and mixed calculations are transformed to floating point when necessary. Almost all the normal built-in arithmetic functions of FORTRAN and PL/I are included. Evaluation is, of course, much slower than if it were carried out by compiled code, but the evaluation capability is much more powerful than that of any other string processor known to the author.

```
STRAN INTERPRETER ON MAY 21,1969 AT 03:05:44.000 PAGE  8


BEGIN READING RULES.


INPUT...(COMPUTE(READ,EVAL))
INPUT...(READ(*INPUT|'#'+$|=*INPUT|2|)END,READ)
INPUT...(EVAL(INPUT|$+';'+$|=OUTPUT|1|#RESULT|1|)END,EVL1)
INPUT...(EVL1(RESULT|$|OUTPUT|$|=*OUTPUT|2+' = '+1|*OUTPUT|' '|)COMPUTE)
INPUT...(COMPUTE)
```

```
BEGIN INTERPRETING RULES.


INPUT...30+50*3;
        30+50*3 = 180

INPUT...(30+50)*3;
        (30+50)*3 = 240

INPUT...(30-50)/4;
        (30-50)/4 = -5

INPUT...-SQRT(4**2+ 3.52**1.3);
        -SQRT(4**2+ 3.52**1.3) = -4.597234E+00

INPUT...FIX(47.1**0.7)MOD 3;
        FIX(47.1**0.7)MOD 3 = 2

INPUT...58.7 MOD 6;
        58.7 MOD 6 = 4.699996E+00

INPUT...SIN(2.0*3.14159265);
        SIN(2.0*3.14159265) = -1.263961E-06

INPUT...9.0**-2;
        9.0**-2 = 1.234567E-02

INPUT...9.0**0.5;
        9.0**0.5 = 2.999998E+00

INPUT...9.0**(3.75/4.86);
        9.0**(3.75/4.86) = 5.448763E+00
```

B.3 Example 3

The problem of translating numbers expressed in the English language, into numeric form is one that has fascinated the author for a long period of time. The solution would appear to be quite simple, but attempts at an actual implementation have always resulted in enough difficulties so that the project has been abandoned. As a subpart of the English language, requiring a grammatical description, it has always seemed an interesting chunk to bite off in an attempt to write an English grammar. The STRAN language allowed a very simple solution. The associative storage is used for the one-one map from English to numeric number names. The rule set is not written as an acceptor, which will reject all sentences not considered acceptable English, but as a translator, which will produce a value for any input. In the case of phrases which denote a number in English in one of the forms which are correctly handled, the rule set will produce the correct numeric form of the number denoted.

The set of rules making up the subprocedure ASSERT, are used to store ordered n-tuples (3-tuples in this case) in the associative storage. This rule set is used as a utility procedure in all applications where information must be stored in the associative memory. Once the English to numeric number name function has been stored, the rule set named TRANSLATE is executed, and data cards are read and transformed to numeric.

BEGIN READING RULES.

```
INPUT...# RULES FOR TRANSLATION OF NUMBERS IN ENGLISH TO NUMERIC.
INPUT...(TRANSLATE(READ,SEMICOLON))
INPUT...(SEMICOLON(INPUT|$+';'+$|=*OUTPUT|' '|*INPUT|1|)END,EVAL)
INPUT...(EVAL(EVLNMB,PRINT))
INPUT...(PRINT(RESULT|$|=*RESULT|1|)TRANSLATE)
INPUT...(EVLNMB(=NUMBER|'0'|RESULT|'C'|)EVN1)
INPUT...(EVN1(INPUT|$+'_ '+$|=INPUT|1+' '+3|)EVN2,EVN1)
INPUT...(EVN2(INPUT|' '+$|=INPUT|2|)EVN3)
INPUT...(EVN3(INPUT|$+' '|=INPUT|1+2|)EVN4,EVSRCH)
INPUT...(EVN4(INPUT|$|=INPUT|1+' '|)EVSRCH)
INPUT...(EVSRCH(INPUT|$+' '+$|=WORD|1|INPUT|3|)EVDONE,EVFND)
INPUT...(EVFND(&F1|'NUMERIC FOR '+$+1|&A1|2|=TEMP|2|)EVSRCH,EVF1)
INPUT...(EVF1(TEMP|$+'00'|)EVF2,EVF3)
INPUT...(EVF2(NUMBER|$|TEMP|$|=NUMBER|1+'+'+2|)EVSRCH)
INPUT...(EVF3(TEMP|$+'000'|)EVF4,EVF5)
INPUT...(EVF4(TEMP|$|NUMBER|$|=NUMBER|1+'*('+2+')'|)EVSRCH)
INPUT...(EVF5(RESULT|$|TEMP|$|NUMBER|$|=NUMBER|'0'|RESULT|1+'+'+2+'*('+3+')'|)EVSRCH)
INPUT...(EVDONE(RESULT|$|NUMBER|$|=*RESULT|1+'+'+2|)END)
INPUT...
INPUT...#  RULES FOR ASSERT.
INPUT...(ASSERT(READ,STORE,ATST))
INPUT...(READ(*INPUT|'#'+$|=*OUTPUT|' '|*INPUT|2|)END,READ)
INPUT...(ATST(INPUT|$+')'+$|)ASSERT,END)
INPUT...(STORE(INPUT|$+'('+$+')'+$|=OUT|3|INPUT|5|)END,STOR)
INPUT...(STOR(S5,STORE))
INPUT...(S5(OUT|$+','+$+','+$+','+$+','+$|)S4,END)
INPUT...(S4(OUT|$+','+$+','+$+','+$|=&S|1+3+5+7|*OUT|'('+1+2+3+4+5+6+7+')'|)S3,END)
INPUT...(S3(OUT|$+','+$+','+$|=&S|1+3+5|*OUT|'('+1+2+3+4+5+')'|)S2,END)
INPUT...(S2(OUT|$+','+$|=&S|1+3|*OUT|'('+1+2+3+')'|)S1,END)
INPUT...(S1(OUT|$|=OUT|1+'))'|)OUT)  THIS RULE ALLOWS EXECUTION OF OTHER ROUTINES
```

BEGIN INTERPRETING RULES.

```
INPUT...(NUMERIC FOR,0,ZERO)(NUMERIC FOR,1,ONE)(NUMERIC FOR,2,TWO)(NUMERIC FOR,3,THREE)
        (NUMERIC FOR,0,ZERO)
        (NUMERIC FOR,1,ONE)
        (NUMERIC FOR,2,TWO)
        (NUMERIC FOR,3,THREE)
INPUT...(NUMERIC FOR,4,FOUR)(NUMERIC FOR,5,FIVE)(NUMERIC FOR,6,SIX)(NUMERIC FOR,7,SEVEN)
        (NUMERIC FOR,4,FOUR)
        (NUMERIC FOR,5,FIVE)
        (NUMERIC FOR,6,SIX)
        (NUMERIC FOR,7,SEVEN)
INPUT...(NUMERIC FOR,8,EIGHT)(NUMERIC FOR,9,NINE)(NUMERIC FOR,10,TEN)
        (NUMERIC FOR,8,EIGHT)
        (NUMERIC FOR,9,NINE)
        (NUMERIC FOR,10,TEN)
INPUT...(NUMERIC FOR,11,ELEVEN)(NUMERIC FOR,12,TWELVE)(NUMERIC FOR,13,THIRTEEN)
        (NUMERIC FOR,11,ELEVEN)
        (NUMERIC FOR,12,TWELVE)
        (NUMERIC FOR,13,THIRTEEN)
INPUT...(NUMERIC FOR,14,FOURTEEN)(NUMERIC FOR,15,FIFTEEN)(NUMERIC FOR,16,SIXTEEN)
        (NUMERIC FOR,14,FOURTEEN)
        (NUMERIC FOR,15,FIFTEEN)
        (NUMERIC FOR,16,SIXTEEN)
INPUT...(NUMERIC FOR,17,SEVENTEEN)(NUMERIC FOR,18,EIGHTEEN)(NUMERIC FOR,19,NINETEEN)
        (NUMERIC FOR,17,SEVENTEEN)
        (NUMERIC FOR,18,EIGHTEEN)
        (NUMERIC FOR,19,NINETEEN)
INPUT...(NUMERIC FOR,20,TWENTY)(NUMERIC FOR,30,THIRTY)(NUMERIC FOR,40,FORTY)
        (NUMERIC FOR,20,TWENTY)
        (NUMERIC FOR,30,THIRTY)
        (NUMERIC FOR,40,FORTY)
```

```
INPUT...(NUMERIC FOR,50,FIFTY)(NUMERIC FOR,60,SIXTY)(NUMERIC FOR,70,SEVENTY)
        (NUMERIC FOR,50,FIFTY)
        (NUMERIC FOR,60,SIXTY)
        (NUMERIC FOR,70,SEVENTY)
INPUT...(NUMERIC FOR,80,EIGHTY)(NUMERIC FOR,90,NINETY)
        (NUMERIC FOR,80,EIGHTY)
        (NUMERIC FOR,90,NINETY)
INPUT...(NUMERIC FOR,100,HUNDRED)
        (NUMERIC FOR,100,HUNDRED)
INPUT...(NUMERIC FOR,1000,THOUSAND)(NUMERIC FOR,1000000,MILLION)
        (NUMERIC FOR,1000,THOUSAND)
        (NUMERIC FOR,1000000,MILLION)
INPUT...(NUMERIC FOR,1000000000,BILLION)
        (NUMERIC FOR,1000000000,BILLION)
INPUT...)))))
```

BEGIN INTERPRETING RULES.

ONE HUNDRED NINETY SEVEN
197

SEVENTY SEVEN MILLION NINE HUNDRED SIXTY THREE THOUSAND FOUR HUNDRED THIRTY TWO
77963432

THIRTY TWO HUNDRED THOUSAND SIXTY NINE HUNDRED AND ONE
3206901

THREE MILLION TWO HUNDRED AND SIX THOUSAND NINE HUNDRED AND ONE
3206901

FORTY FIVE HUNDRED AND FIFTY SIX
4556

B.4 Example 4

Victor Yngve (the developer of COMIT), who used to head the Mechanical Translation group of the Research Laboratory of Electronics, used a program which randomly generated English sentences from a (discontinuous) phrase structure grammar to demonstrate the use of the COMIT language in linguistics. The grammar used was derived from ten sentences from one of his small son's books, which was concerned with trains and an engineer named Small. The output was interesting both because it demonstrated that the grammar was correct, i.e. the sentences were all syntactically correct English, but also because it demonstrated how important semantic content is, and how desperately one will try to find some meaning in a sentence which is structurally correct gibberish. A disadvantage of Yngve's program was that the grammar had to be encoded into the COMIT language, each grammar rule becoming a COMIT rule of a form that was not particularly natural. The program which is presented below uses the same grammar, but here the grammar is stored as data (2-tuples) in the hash coded associative memory. As a result, the form of the grammar is very simple, and the program could be used to test different phrase structure grammars created by linguists who know no programming. Another feature of the program is a pseudo-random number generator written as a single rule (R3).

BEGIN READING RLES.

```
INPUT...# RULES TO GENERATE NONSENSE SENTENCES GIVEN A PHRASE STRUCTURE GRAMMAR.
INPUT...# THE PHRASE STRUCTURE GRAMMAR IS STORED IN THE FACT STORAGE AS 2-TUPLES.
INPUT...(GENERATE(=NUMB|'1'|SV|'37654879'|*CUT|'BEGIN GENERATION OF SENTENCES.'|)LOOP)
INPUT...(LOOP(NUMB|$|=PD|'1,'|*CUT|' '|CUT|'('+1+') '|)NEXT)
INPUT...(NEXT(PD|$+',','+$|=NM|1|PD|3|)DONE,GET)
INPUT...(GET(NM|$|&F1|1+$|&A1|2|=RL|2|)ERR,RL)
INPUT...(R1(RL|$+')')'+$+',','+$|PD|$|&F1|3+$|&A1|7|=NM|3|RL|7|PD|5+4+6|)ERR,RL)
INPUT...(R2(RL|$+'))'+$+',','+$|PD|$+',','+$|&F1|3+$|&A1|9|=NM|3|RL|9|PD|6+7+5+7+8|)ERR,RL)
INPUT...(R3(SV|$|RL|$+'))'+$|=#RL|'1+('+1+'/333)MOD '+4|#SV|1+'*17357|)ERR,R3A)
INPUT...(R3A(NM|$|RL|$|=NM|1+':'+2|)GET)
INPUT...(R4(CUT|$65+$|RL|$+')) '+$|=RL|'R4B,R4C'+4+5|)R4A,RL)
INPUT...(R4A(OUT|$|RL|$+'))'+$|=CUT|1+4|)ERR,NEXT)
INPUT...(R4B(OUT|$+'* '+$|=OUT|1+'*'+3|)END,R4C)
INPUT...(R4C(CUT|$+'**'+$|=CUT|1+'*'+3|)END)
INPUT...(R4D(OUT|$|RL|$+'))'+$|=*CUT|1|CUT|'    '+4|)ERR,NEXT)
INPUT...(R5(RL|$+')')'+$|&F1|3+$|&A1|4|=NM|3|RL|4|)ERR,RL)
INPUT...(DONE(=RL|'R4B,DPRT))'|)RL)
INPUT...(ERR(NM|$|PD|$|=*PD|'*ERROR* NM='+1+' PD='+2|)DPRT)
INPUT...(DPRT(NUMB|$|CUT|$|=#NUMB|'1+'+1|*CUT|2|)DTST)
INPUT...(DTST(NUMB|'11'|)LOOP,END)
INPUT...
INPUT...# RULES FOR ASSERT.  THIS IS A SPECIAL VERSION TO STORE C.F. GRAMMAR RULES.
INPUT...(ASSERT(READ,STORE,ATST))
INPUT...(READ(*INPUT|'#'+$|)END,READ)
INPUT...(ATST(INPUT|$+')'+$|)ASSERT,END)
INPUT...(STORE(INPUT|$+'('+$+')'+$|=OUT|3|INPUT|5|)END,STOR)
INPUT...(STOR(S5,STORE))
INPUT...(S5(CUT|$+'='+$|=N1|1|N2|'R5))'+3|)S1,S4)
INPUT...(S4(N2|'R5))'+$+'+'+$|=N2|'R1))'+2+',+4|)S3,S0)
INPUT...(S3(N2|'R5))'+$+'...'+$|=N2|'R2))'+2+',+4|)S2,S0)
```

```
INPUT...(S2(N2|'R5))<'+$+'>'+$|=N2|'R4))'+2|)SC)
INPUT...(S1(OUT|$+'*'+$|=N1|1|N2|'R3))'+3|)SERR,SO)
INPUT...(SO(N1|$|N2|$|=&S|1+2|)END)
INPUT...(SERR(OUT|$|=*OUT|'('+1+') IS AN ILLEGAL RULE.'|)END)
INPUT...(ASSERT)
```

```
STRAN INTERPRETER ON MAY 20,1969 AT 18:23:38.320 PAGE 10


BEGIN INTERPRETING RULES.


INPUT...# GRAMMAR OF TEN ENGLISH SENTENCES. STORED AS 2-TUPLES BY ASSERT.
INPUT...# THIS IS A MODIFIED GRAMMAR TO HANDLE QUESTIONS.
INPUT...(1*2)(1:1=2)(1:2=109)(2=3+7)(3=4)(4=5...6)(5=<*>)(6=<.>)(7*2)(7:1=8)(7:2=17)
INPUT...(8=9+16)(9*2)(9:1=10)(9:2=15)(10=11+14)(11=12)(12=13+16)(13*2)(13:1=< WHEN>)
INPUT...(13:2=< IF>)(14=<,>)(15=16+13)(16=17)(17=60+18)(18*2)(18:1=21)(18:2=22)(19=<?>)
INPUT...(21=23+38)(22=24+62)
INPUT...(23=< IS>)(24*5)(24:1=< HAS>)(24:2=< KEEPS>)(24:3=< MAKES>)(24:4=28)
INPUT...(24:5=35)(28=29...31)(29=< HAS>)(31=32)(32=33+62)(33*2)(33:1=< IN>)
INPUT...(33:2=< UNDER>)(35=36...38)(36*2)(36:1=< KEEPS>)(36:2=< MAKES>)(38*3)
INPUT...(38:1=55)(38:2=40)
INPUT...(38:3=47)(40=47+41)(41*2)(41:1=45)(41:2=43)(43=44+46.1)(44=<,>)
INPUT...(45=46+47)(46=< AND>)(46.1=40)(47*8)(47:1=< BLACK>)(47:2=< SHINY>)
INPUT...(47:3=< OILED>)(47:4=< POLISHED>)(47:5=< HEATED>)(47:6=< BIG>)
INPUT...(47:7=< LITTLE>)(47:8=< PROUD>)(55=56+57)(56=< PROUD>)(57=58)(58=59+62)
INPUT...(59=< OF>)(60*2)(60:1=< HE>)(60:2=71)(62*2)(62:1=69)(62:2=64)(64=69+65)
INPUT...(65*2)(65:1=67)(65:2=68)(67=44+68.1)(68=46+69)(68.1=64)(69*2)(69:1=71)
INPUT...(69:2=93)(71*7)(71:1=79)(71:2=77)(71:3=< IT>)(71:4=82)(71:5=80)(71:6=83)
INPUT...(71:7=84)(77=78+79)(78=< *ENGINEER>)(79=< *SMALL>)(80*2)(80:1=< WATER>)
INPUT...(80:2=< STEAM>)(82=85+80)(83=85+103)(84=89+103)(85*3)(85:1=88)(85:2=92)
INPUT...(85:3=90)(88=< THE>)(89=< A>)(90*2)(90:1=< HIS>)(90:2=< ITS>)(92=88...31)
INPUT...(93*3)(93:1=95)(93:2=100)(93:3=98)(95=85+96)(96*2)(96:1=100)(96:2=98)
INPUT...(98=102+99)(99=100)(100=103+101)(101=<S>)(102=< FOUR>)(103*2)(103:1=108)
INPUT...(103:2=105)(105=106+108)(106*2)(106:1=47)(106:2=40)(108*11)
INPUT...(108:1=< DRIVING WHEEL>)(108:2=< TRAIN>)(108:3=< ENGINE>)(108:4=< BELL>)
INPUT...(108:5=< WHISTLE>)(108:6=< SAND DOME>)(108:7=< HEADLIGHT>)
INPUT...(108:8=< SMOKESTACK>)(108:9=< WHEEL>)(108:10=< FIRE BOX>)(108:11=< BOILER>)
INPUT...(109=110+112)(110=111)(111=5...19) (112*3)(112:1=113)(112:2=116)(112:3=118)
INPUT...(113=114+60)(114=115)(115=23...38)(116=10+113)(118=113+11)
INPUT...# END OF GRAMMAR.
INPUT...))))))))))
```

BEGIN INTERPRETING RULES.

BEGIN GENERATION OF SENTENCES.

(1) *IS STEAM POLISHED AND BLACK?

(2) *IF HE MAKES FOUR PROUD SMOKESTACKS, IS WATER PROUD OF THE STEAM AND THE TRAIN?

(3) *HE IS HEATED.

(4) *IS THE BELL IN THE WATER PROUD OF WATER, POLISHED WHEELS AND OILED TRAINS IF HE KEEPS FOUR BLACK SMOKESTACKS?

(5) *HE IS PROUD AND HEATED.

(6) *HE MAKES THE OILED SMOKESTACKS LITTLE, LITTLE, POLISHED, BIG AND BIG IF A POLISHED DRIVING WHEEL KEEPS ITS STEAM.

(7) *A BLACK, BIG, OILED, PROUD AND OILED BELL MAKES THE DRIVING WHEELS.

(8) *IT KEEPS THE PROUD AND HEATED TRAIN UNDER HIS POLISHED, HEATED, BLACK AND BIG FIRE BOX AND SMOKESTACKS.

(9) *HE MAKES IT, ITS SMOKESTACK AND THE WATER IF IT IS BIG, OILED, OILED, BIG AND BLACK.

(10) *HE KEEPS THE DRIVING WHEELS PROUD OF IT.

B.5 Example 5

This example is the first that is directly relevant to COMS. The two rule sets to be demonstrated here carry out the elementary operations for a fact retrieval system using the Set Theoretic Language. The two operations, ASSERT and ANSWER, implement the storage and retrieval of n-tuples ($2 \leq n \leq 4$) in the hash-coded pseudo-associative memory of the STRAN processor.

ASSERT stores closed n-tuples in the memory. If the n-tuple being asserted has not previously been stored, the n-tuple will be stored in memory and printed in the output. If the n-tuple is already stored, no action is taken.

ANSWER looks for answers to closed and open n-tuples. The answer to a closed n-tuple is either "TRUE" or "DON'T KNOW". The answer to an open n-tuple (one which has positions where any symbol will be accepted) is either a statement that the question has no answers, or a list of the answers found.

This example, and examples 6 and 7 which follow, form a complete run of the STRAN interpreter. Facts, in the form of sentences of STL, are stored in this example. In example 6, more facts are stored, but the input is in the form of simple English sentences. Finally in example 7, deductions are carried out on the facts stored previously. All deduced facts not previously stored are printed as output and stored in the associative memory.

BEGIN READING RULES.

```
INPUT...# RULES FOR ANSWER. USES RULES FROM ASSERT.
INPUT...(ANSWER(READ,FIND,NTST))
INPUT...(NTST(INPUT|$+')'+$|)ANSWER,END)
INPUT...(FIND(INPUT|$+'('+$+')'+$|=*OUT|' '|OUT|3|INPUT|5|)END,FND)
INPUT...(FND(F5,FIND))
INPUT...(F5(OUT|$+','+$+','+$+','+$+','+$|)F4,END)
INPUT...(F4(OUT|$+','+$+','+$+','+$|)F3,AN4)
INPUT...(F3(OUT|$+','+$+','+$|)F2,AN3)
INPUT...(F2(OUT|$+','+$|)S1,AN2)
INPUT...(AN4(&F1|1+3+5+7|)FAIL,FOUND)
INPUT...(AN3(&F1|1+3+5|)FAIL,FOUND)
INPUT...(AN2(&F1|1+3|)FAIL,FOUND)
INPUT...(FAIL(OUT|$+'$'+$|=*OUT|'THERE ARE NO ANSWERS TO ('+1+2+3+').'|)FCLSD,END)
INPUT...(FCLSD(OUT|$|=*OUT|'THE TRUTH OR FALSEHOOD OF ('+1+') IS UNKNOWN.'|)END)
INPUT...(FOUND(OUT|$+'$'+$|=*OUTPUT|'('+1+2+3+') HAS THESE ANSWERS:'|)CLOSED,FND3)
INPUT...(FND3(OUT|$+'$'+$+'$'+$+'$'+$|)FND2,AC3)
INPUT...(FND2(OUT|$+'$'+$+'$'+$|)FND1,AC2)
INPUT...(FND1(OUT|$+'$'+$|)CLOSED,AC1)
INPUT...(AC3(&A1|2+4+6|=*OUT|' ('+1+2+3+4+5+6+7+')'|)END,AC3)
INPUT...(AC2(&A1|2+4|=*OUT|'('+1+2+3+4+5+')'|)END,AC2)
INPUT...(AC1(&A1|2|=*OUT|'('+1+2+3+')'|)END,AC1)
INPUT...(CLOSED(OUT|$|=*OUT|'('+1+') IS TRUE.'|)END)
INPUT...(END)
```

BEGIN READING RULES.

```
INPUT...#  RULES FOR ASSERT.
INPUT...(ASSERT(READ,STORE,ATST))
INPUT...(READ(*INPUT|'#'+$|=*OUTPUT|' '|*INPUT|2|)END,READ)
INPUT...(ATST(INPUT|$+')'+$|)ASSERT,END)
INPUT...(STORE(INPUT|$+'('+$+')'+$|=OUT|3|INPUT|5|)END,STOR)
INPUT...(STOR(S5,STORE))
INPUT...(S5(OUT|$+','+$+','+$+','+$+','+$|)S4,END)
INPUT...(S4(OUT|$+','+$+','+$+','+$|=&S|1+3+5+7|*OUT|'('+1+2+3+4+5+6+7+')'|)S3,END)
INPUT...(S3(OUT|$+','+$+','+$|=&S|1+3+5|*OUT|'('+1+2+3+4+5+')'|)S2,END)
INPUT...(S2(OUT|$+','+$|=&S|1+3|*OUT|'('+1+2+3+')'|)S1,END)
INPUT...(S1(OUT|$|=OUT|1+'))'|)OUT)  THIS RULE ALLOWS EXECUTION OF OTHER ROUTINES
INPUT...(NOECHO)
```

STRAN INTERPRETER ON MAY 21,1969 AT 03:05:45.750 PAGE 14

BEGIN INTERPRETING RULES.

```
 SET THEORETIC MODEL OF PRIMITIVE RELATIONS USED BY DEDUCE.
(INVERSE OF,INVERSE OF,INVERSE OF)
(INVERSE OF,DISJOINT FROM,DISJOINT FROM)
(INVERSE OF,IDENTICAL TO,IDENTICAL TO)
(CHAIN OF&AND,DISJOINT FROM,SUBSET OF,DISJOINT FROM)
(CHAIN OF&AND,SUBSET OF,SUBSET OF,SUBSET OF)
(CHAIN OF&AND,COMES FROM,COMES FROM,IN)
(CHAIN OF&AND,IN,IN,IN)

 SET THEORETIC LANGUAGE MODEL OF HUMAN RELATIONS.
(INVERSE OF,PARENT OF,OFFSPRING OF)
(INVERSE OF,SIBLING OF,SIBLING OF)
(INVERSE OF,MARRIED TO,MARRIED TO)
(INVERSE OF,HUSBAND OF,WIFE OF)
(INVERSE OF,COUSIN OF,COUSIN OF)
(CHAIN OF&AND,GRANDPARENT OF,PARENT OF,PARENT OF)
(CHAIN OF&AND,GRANDFATHER OF,FATHER OF,PARENT OF)
(CHAIN OF&AND,GRANDMOTHER OF,MOTHER OF,PARENT OF)
(CHAIN OF&AND,GRANDSON OF,SON OF,OFFSPRING OF)
(CHAIN OF&AND,GRANDDAUGHTER OF,DAUGHTER OF,OFFSPRING OF)
(CHAIN OF&AND,GRANDCHILD OF,CHILD OF,OFFSPRING OF)
(CHAIN OF&AND,UNCLE OF,BROTHER OF,PARENT OF)
(CHAIN OF&AND,AUNT OF,SISTER OF,PARENT OF)
(CHAIN OF&AND,NIECE OF,DAUGHTER OF,SIBLING OF)
(CHAIN OF&AND,NEPHEW OF,SON OF,SIBLING OF)
(CHAIN OF&AND,COUSIN OF,OFFSPRING OF,UNCLE OF)
(CHAIN OF&AND,COUSIN OF,OFFSPRING OF,AUNT OF)
(CHAIN OF&AND,COUSIN OF,NEPHEW OF,PARENT OF)
(CHAIN OF&AND,COUSIN OF,NIECE OF,PARENT OF)
```

```
(CHAIN OF&AND,CCUSIN CF,SIBLING CF,CCUSIN CF)
(CHAIN CF&ANC,MOTHER IN LAW OF,MOTHER OF,MARRIED TO)
(CHAIN CF&ANC,FATHER IN LAW OF,FATHER CF,MARRIED TO)
(CHAIN OF&AND,BROTHER IN LAW CF,HUSBAND CF,SIBLING CF)
(CHAIN CF&ANC,BROTHER IN LAW OF,BROTHER OF,MARRIED TO)
(CHAIN OF&AND,SISTER IN LAW CF,WIFE CF,SIBLING CF)
(CHAIN OF&AND,SISTER IN LAW CF,SISTER OF,MARRIED TO)
(CHAIN CF&ANC,SON IN LAW CF,HUSBAND OF,DAUGHTER OF)
(CHAIN OF&AND,DAUGHTER IN LAW CF,WIFE CF,SON CF)
(UNION CF&ANC,MARRIED,HUSBAND,WIFE)
(UNION CF&ANC,PARENT,FATHER,MOTHER)
(UNION OF&AND,GRANDPARENT,GRANDFATHER,GRANDMOTHER)
(UNION CF&ANC,OFFSPRING,SON,DAUGHTER)
(UNION OF&AND,SIBLING,BROTHER,SISTER)
(UNION OF&AND,PERSON,MALE PERSON,FEMALE PERSON)
(UNION OF&ANC,CHILD,BOY,GIRL)
(INTERSECTION OF&AND,MAN,ADULT,MALE PERSON)
(INTERSECTION OF&AND,WOMAN,ADULT,FEMALE PERSON)
(INTERSECTION CF&ANC,BACHELOR,UNMARRIED,MAN)
(INTERSECTION OF&AND,BOY,CHILD,MALE PERSON)
(INTERSECTION OF&ANC,GIRL,CHILD,FEMALE PERSON)
(INTERSECTION OF&AND,FEMALE PERSON,FEMALE,PERSON)
(INTERSECTION OF&AND,MALE PERSON,MALE,PERSON)
(LEFT INTERSECTION CF&ANC,CHILD CF,OFFSPRING OF,CHILD)
(LEFT INTERSECTION OF&AND,WIFE CF,MARRIED TO,WOMAN)
(LEFT INTERSECTION OF&ANC,HUSBAND OF,MARRIED TO,MAN)
(LEFT INTERSECTION OF&ANC,SON CF,OFFSPRING CF,MALE)
(LEFT INTERSECTION OF&AND,DAUGHTER CF,OFFSPRING CF,FEMALE)
(LEFT INTERSECTION CF&ANC,FATHER CF,PARENT CF,MAN)
(LEFT INTERSECTION CF&ANC,MOTHER CF,PARENT CF,WOMAN)
(LEFT INTERSECTION OF&ANC,SISTER CF,SIBLING CF,FEMALE)
(LEFT INTERSECTION CF&ANC,BROTHER CF,SIBLING CF,MALE)
(LEFT INTERSECTION OF&AND,GRANDFATHER CF,GRANDPARENT CF,MAN)
```

```
(LEFT INTERSECTION OF&AND,GRANDMOTHER OF,GRANDPARENT OF,WOMAN)
(LEFT HALF OF,FATHER,FATHER OF)
(LEFT HALF OF,MOTHER,MOTHER OF)
(LEFT HALF OF,SON,SON OF)
(LEFT HALF OF,DAUGHTER,DAUGHTER OF)
(LEFT HALF OF,PARENT,PARENT OF)
(LEFT HALF OF,CHILD,CHILD OF)
(LEFT HALF OF,UNCLE,UNCLE OF)
(LEFT HALF OF,AUNT,AUNT OF)
(LEFT HALF OF,COUSIN,COUSIN OF)
(LEFT HALF OF,BROTHER,BROTHER OF)
(LEFT HALF OF,SISTER,SISTER OF)
(LEFT HALF OF,HUSBAND,HUSBAND OF)
(LEFT HALF OF,WIFE,WIFE OF)
(LEFT HALF OF,SIBLING,SIBLING OF)
(LEFT HALF OF,GRANDFATHER,GRANDFATHER OF)
(LEFT HALF OF,GRANDDAUGHTER,GRANDDAUGHTER OF)
(LEFT HALF OF,GRANDMOTHER,GRANDMOTHER OF)
(LEFT HALF OF,GRANDSON,GRANDSON OF)
(LEFT HALF OF,MARRIED,MARRIED TO)
(LEFT HALF OF,OFFSPRING,OFFSPRING OF)
(LEFT HALF OF,MOTHER IN LAW,MOTHER IN LAW OF)
(LEFT HALF OF,FATHER IN LAW,FATHER IN LAW OF)
(LEFT HALF OF,BROTHER IN LAW,BROTHER IN LAW OF)
(LEFT HALF OF,SISTER IN LAW,SISTER IN LAW OF)
(LEFT HALF OF,SON IN LAW,SON IN LAW OF)
(LEFT HALF OF,DAUGHTER IN LAW,DAUGHTER IN LAW OF)
(RIGHT HALF OF,OFFSPRING,MOTHER OF)
(RIGHT HALF OF,OFFSPRING,FATHER OF)
(SUBSET OF,PERSON,OFFSPRING)
(SUBSET OF,FATHER,MAN)
(SUBSET OF,MOTHER,WOMAN)
(SUBSET OF,HUSBAND,MAN)
```

```
(SUBSET OF,WIFE,WOMAN)
(SUBSET OF,GRANDPARENT,PARENT)
(SUBSET OF,PARENT,ADULT)
(SUBSET OF,CHILD,OFFSPRING)
(SUBSET OF,CHILD,UNMARRIED)
(SUBSET OF,MARRIED,PERSON)
(SUBSET OF,MARRIED,ADULT)
(SUBSET OF,CHILD,PERSON)
(SUBSET OF,MAN,PERSON)
(SUBSET OF,WOMAN,PERSON)
(DISJOINT FROM,PLANT,ANIMAL)
(DISJOINT FROM,PERSON,PLANT)
(DISJOINT FROM,UNMARRIED,MARRIED)
(DISJOINT FROM,MALE,FEMALE)
(DISJOINT FROM,CHILD,ADULT)
```

BEGIN INTERPRETING RULES.

INPUT... THE FOLLOWING ARE QUESTIONS INPUT TO ANSWER.
INPUT...(DISJOINT FROM,PLANT,ANIMAL)(BOY,ERIC)(INVERSE OF,$,$)($,CHILD,$)($,TREE,$))

(DISJOINT FROM,PLANT,ANIMAL) IS TRUE.

THE TRUTH OR FALSEHOOD OF (BOY,ERIC) IS UNKNOWN.

(INVERSE OF,$,$) HAS THESE ANSWERS:
(INVERSE OF,INVERSE OF,INVERSE OF)
(INVERSE OF,OFFSPRING OF,PARENT OF)
(INVERSE OF,WIFE OF,HUSBAND OF)
(INVERSE OF,COUSIN OF,COUSIN OF)
(INVERSE OF,HUSBAND OF,WIFE OF)
(INVERSE OF,MARRIED TO,MARRIED TO)
(INVERSE OF,SIBLING OF,SIBLING OF)
(INVERSE OF,PARENT OF,OFFSPRING OF)
(INVERSE OF,IDENTICAL TO,IDENTICAL TO)
(INVERSE OF,DISJOINT FROM,DISJOINT FROM)

($,CHILD,$) HAS THESE ANSWERS:
(LEFT HALF OF,CHILD,CHILD OF)
(DISJOINT FROM,CHILD,BACHELOR)
(DISJOINT FROM,CHILD,GRANDFATHER)
(DISJOINT FROM,CHILD,GRANDMOTHER)
(DISJOINT FROM,CHILD,WIFE)
(DISJOINT FROM,CHILD,HUSBAND)
(DISJOINT FROM,CHILD,WOMAN)
(DISJOINT FROM,CHILD,MAN)
(DISJOINT FROM,CHILD,GRANDPARENT)
(DISJOINT FROM,CHILD,PARENT)

```
(DISJOINT FROM,CHILD,MOTHER)
(DISJOINT FROM,CHILD,FATHER)
(DISJOINT FROM,CHILD,MARRIED)
(DISJOINT FROM,CHILD,PLANT)
(DISJOINT FROM,CHILD,ADULT)
(SUBSET OF,CHILD,PERSON)
(SUBSET OF,CHILD,UNMARRIED)
(SUBSET OF,CHILD,OFFSPRING)

THERE ARE NO ANSWERS TO ($,TREE,$).
```

B.6 Example 6

While working with the Set Theoretic Language, it was noticed that if the symbols used in the n-tuples were correctly formed, a relatively simple algorithm could be described for transforming any n-tuple into at least one simple English sentence. As a result, a rule set has been created to accept simple English sentences (of a rather restricted type) and transform them into facts encoded as sentences of STL. For most sentences a single n-tuple is produced to represent the meaning of the sentence, but extra n-tuples are also produced which give the syntactic categories of the symbols making up that n-tuple. An important characteristic of the rule set is that it knows only a few simple structural words of English, and has no need for knowing the syntactic categories of all the words in a sentence. Thus, sentences can be parsed which contain many words which are completely unknown to the parser. This allows facts about strange subject matter to be introduced, without the syntax of the terminology having to be completely predigested for the parser. Thus, the parser allows diverse facts written in English to be stored in the pseudo-associative memory, just as if they had been encoded in STL and stored by ASSERT (see example 5 of this appendix). Not only can the parser handle simple facts about the problem space, but it can deal with abstract facts about relations among the primitive relations. For example, the sentence, "The relation 'inverse of' is the inverse of itself." is equivalent to the sentence of STL, (inverse of, inverse of, inverse of). In the following we present the parsing

of English sentences in a number of different subject areas. It is the hope of the author that the subset of English handled is clear enough to be learned by seeing a number of examples. If that is true, the subset should be able to be learned in exactly the manner that an interested adult learns the subset of English known by a young child. If that is indeed possible, then untrained people should be able to learn to communicate with this fact retrieval system by using it.

In the printed output shown, the first line of each set of statements is the input English sentence. The succeeding statements are sentences of the Set Theoretic Language which resulted from the parsing, and were not previously stored in the memory.

BEGIN READING RULES.

```
INPUT...#  RULES FOR PARSE. STORES RESULTS IN ASSOCIATIVE MEMORY.  USES ASSERT.
INPUT...(PARSE(READ,BEGIN,STORE,PTEST))
INPUT...(PTEST(INPUT|$+')'+$|)PARSE,END)
INPUT...(BEGIN(INPUT|$+'.  '+$|=*OUTPUT|' '|*SENTENCE|1+'.'|INPUT|3|)BEGUN,B1)
INPUT...(BEGUN(INPUT|$+'*'+$+'.'+$|=*OUTPUT|' '|*SENTENCE|2+3+4|INPUT|5|)END,DBL)
INPUT...(B1(SENTENCE|' '+$|=SENTENCE|2|)DBL,B1)
INPUT...(DBL(SENTENCE|$+'  '+$|=SENTENCE|1+' '+3|)RULE1,DBL)
INPUT...(RULE1(SENTENCE|$+' IS '+$+'.'|=NP|1+' '|VP|' '+3|)ACTV,VPD1)
INPUT...(VPD1(VP|DETS+$|=VP|' '+2|VPTYP|'NOUN,'|)VPD3,RULE2)
INPUT...(DETS(' A ' AN '))
INPUT...(VPD3(VP|' THE '+$|=VP|' '+2|VPTYP|'FUNCTION,'|)VPD4,RULE2)
INPUT...(VPD4(=VPTYP|'ADJECTIVE,'|)RULE2)
INPUT...(RULE2(NP|'*HE '+$|SUBJ|$|=OUT|'MALE,'+3|TMP|'S2,PREPS))'|)RULE3,TMP)
INPUT...(RULE3(NP|'*SHE '+$|SUBJ|$|=OUT|'FEMALE,'+3|TMP|'S2,PREPS))'|)RULE4,TMP)
INPUT...(RULE4(NP|'*IT '+$|SUBJ|$|=OUT|'NEUTER,'+3|TMP|'S2,PREPS))'|)RULE5,TMP)
INPUT...(RULE5(NP|D1+$+' '|=OUT|'NOUN,'+2|SUBJ|2|TMP|'S2,RSUBS))'|)RULE6,TMP)
INPUT...(D1('*ANY '*EVERY '))
INPUT...(RULE6(NP|D2+$+' '|=OUT|'ADJECTIVE,'+2|SUBJ|2|TMP|'S2,RSUBS))'|)RULE7,TMP)
INPUT...(D2('*ANYTHING '*EVERYTHING '))
INPUT...(RULE7(NP|'*NO '+$+' '|=OUT|'NOUN,'+2|SUBJ|2|TMP|'S2,RDISJ))'|)RULE8,TMP)
INPUT...(RULE8(NP|'*NOTHING '+$+' '|=OUT|'ADJECTIVE,'+2|SUBJ|2|)RULE9,RULE8A)
INPUT...(RULE8A(=TMP|'S2,RDISJ))'|)TMP)
INPUT...(RULE9(NP|$+PREF+$+' '|=SUBJ|3|)GSUBJ,PREPS)
INPUT...(GSUBJ(NP|$+' '|=SUBJ|1|)PREPS)
INPUT...(RSUBS(SUBJ|$|VP|' '+$|=RSS|'(SUBSET OF,'+1+','+3+')'|)RVPT)
INPUT...(RDISJ(SUBJ|$|VP|' '+$|=RSS|'(DISJOINT FROM,'+1+','+3+')'|)RVPT)
INPUT...(PREPS(VP|$+PREPOSITNS+$|=RELTN|1+2|OBJ|3|)TEST,3T3)
INPUT...(PREPOSITNS(' OF ' IN ' TO ' THAN ' FROM ' AT ' FOR ' BETWEEN '))
INPUT...(2TUP(VP|' '+$+','+$|=VP|4|)2T1,2TA)
INPUT...(2T1(VP|' AND '+$|=VP|' '+2|)2T2,2T3)
```

```
INPUT...(2T2(VP|' '+$+' '+$|=VP|' '+4|)2T3,2TA)
INPUT...(2T3(VP|' '+$|SUBJ|$|=RSS|'('+2+','+3+')'|)RVPT)
INPUT...(2TA(SUBJ|$|=OUT|'ADJECTIVE,'+2|RSS|2+',"+1|TMP|'S2,2TB,S2,2TUP))'|)TMP)
INPUT...(2TB(RSS|$|=CUT|1|)END)
INPUT...(SPSHL('S OF 'S IN 'S TC 'S FRCM 'S AT 'S FCR '))
INPUT...(ACTV(SENTENCE|$+' '+$+SPSHL+$+'.'|=NP|1+2|CUTPUT|3|VP|4+5|)UNABLE,A1)
INPUT...(A1(NP|$|CUTPUT|$+' '+$|=NP|1+2+' '|CUTPUT|4|)A2,A1)
INPUT...(A2(OUTPUT|$|VP|$|=VP|' '+1+2|VPTYP|'VERB PHRASE,'|)RULE2)
INPUT...(PREF('THE RELATICN 'THE PRCPERTY 'THE WORD '))
INPUT...(ALST(' AND THE PRCPERTY ' AND THE RELATICN '))
INPUT...(3T3(OBJ|PREF+$|=OBJ|2|)3T4)
INPUT...(3T4(CBJ|'ITSELF'+$|SUBJ|$|=CBJ|3+2|)3T5)
INPUT...(3T5(CBJ|$+' AND ITSELF'|SUBJ|$|RELTN|$+' '|=CBJ|1+',"+3|RELTN|4+'&AND '|)3T6)
INPUT...(3T6(OBJ|$+ALST+$|RELTN|$+' '|=OBJ|1+',"+3|RELTN|4+'&AND '|)3T7)
INPUT...(3T7(OBJ|'THE '+$|=CUT|'UNIQUE,'+2|CBJ|2|TMP|'S2,3T8))'|)3T8,TMP)
INPUT...(3T8(OBJ|'*'+$|=OUT|'PROPER NAME,'+1+2|TMP|'S2,3TUP))'|)3T9,TMP)
INPUT...(3T9(CBJ|$+PREPCSITNS+$|=CUTPUT|2|CBJ|1+',"+3|)3TUP,3TN)
INPUT...(3TN(OUTPUT|' '+$|RELTN|$+' '|=RELTN|3+'&'+2|)3TUP)
INPUT...(3TUP(VPTYP|'NOUN,'|RELTN|' '+$+' '|=OUT|'NOUN RCOT,'+3|TMP|'S2,3TF))'|)3T1,TMP)
INPUT...(3T1(VPTYP|'VERB'+$|RELTN|' '+$+' '|=OUT|'VERB ROOT,'+4|TMP|'S2,3TF))'|)3T2,TMP)
INPUT...(3T2(VPTYP|'FUNCTION,'|RELTN|' '+$+' '|=CUT|1+3|TMP|'S2,SUPER))'|)3TF,TMP)
INPUT...(SUPER(RELTN|' '+$+' '|=OUT|'NOUN ROOT,'+2|TMP|'S2,3TF))'|)3TF,TMP)
INPUT...(3TF(RELTN|' '+$+' '|SUBJ|$|CBJ|$|=RSS|'('+2+','+4+','+5+')'|)PNAM)
INPUT...(TEST(SUBJ|'*THE '+$|VP|' '+$|=SUBJ|4|VP|' '+2|VPTYP|'FUNCTICN,'|)2TUP,PREPS)
INPUT...(RVPT(VPTYP|'AD'+$|VP|' '+$+' '+$|=OUT|'ADJECTIVE PHRASE,'+4+5+6|)RVP1,RVPA)
INPUT...(RVPA(=TMP|'S2,PNAM))'|)TMP)
INPUT...(RVP1(VPTYP|$|VP|' '+$|=OUT|1+3|TMP|'S2,PNAM))'|)PNAM,TMP)
INPUT...(PNAM(SUBJ|'*'+$|=CUT|'PROPER NAME,'+1+2|TMP|'S2,PRINT))'|)PRINT,TMP)
INPUT...(PRINT(RSS|$+'('+$+')'+$|=CUT|3|TMP|'S5,BEGIN))'|)BEGIN,TMP)
INPUT...(UNABLE(SENTENCE|$|=*CUTPUT|'UNABLE TC PARSE '+1|)BEGIN)
INPUT...(NOECHO)
```

BEGIN INTERPRETING RULES.

ENGLISH LANGUAGE MODEL OF NORMAN'S FAMILY.

```
*NORMAN IS THE FATHER OF *ERIC.
(PROPER NAME,*ERIC)
(FUNCTION,FATHER OF)
(NOUN ROOT,FATHER OF)
(PROPER NAME,*NORMAN)
(FATHER OF,*NORMAN,*ERIC)

*MARVIN IS THE FATHER OF *NORMAN.
(PROPER NAME,*MARVIN)
(FATHER OF,*MARVIN,*NORMAN)

*PAT IS A DAUGHTER OF *ALICE.
(PROPER NAME,*ALICE)
(NOUN ROOT,DAUGHTER OF)
(PROPER NAME,*PAT)
(DAUGHTER OF,*PAT,*ALICE)

*ERIC IS A LITTLE BOY.
(ADJECTIVE,LITTLE)
(LITTLE,*ERIC)
(NOUN,BOY)
(BOY,*ERIC)

*BARTON IS A BROTHER OF *NORMAN.
(NOUN ROOT,BROTHER OF)
(PROPER NAME,*BARTON)
(BROTHER OF,*BARTON,*NORMAN)
```

```
*MARVIN IS MARRIED TO *ALICE.
(MARRIED TO,*MARVIN,*ALICE)

*NORMAN IS MARRIED TO *MADELAINE.
(PROPER NAME,*MADELAINE)
(MARRIED TO,*NORMAN,*MADELAINE)

*BARTON IS MARRIED TO *MERLA.
(PROPER NAME,*MERLA)
(MARRIED TO,*BARTON,*MERLA)

 *KEVIN IS A SON OF *BARTON.
(NOUN ROOT,SON OF)
(PROPER NAME,*KEVIN)
(SON OF,*KEVIN,*BARTON)

*CHRISTOPHER IS A SON OF *NORMAN.
(PROPER NAME,*CHRISTOPHER)
(SON OF,*CHRISTOPHER,*NORMAN)

*HE IS A CHILD.
(MALE,*CHRISTOPHER)
(NOUN,CHILD)
(CHILD,*CHRISTOPHER)

*MANDRES IS IN *FRANCE.
(PROPER NAME,*FRANCE)
(PROPER NAME,*MANDRES)
(IN,*MANDRES,*FRANCE)

*MADELAINE COMES FROM *MANDRES.
(VERB ROOT,COMES FROM)
```

B.7 Example 7

A collection of n-tuples can form a description of a problem space. However, to do this they must be provided with a syntax and a semantics, either explicit or understood. The syntax we have provided is that of the Set Theoretic Language. If an n-tuple has satisfactory form under the criteria of STL, then it can be called a sentence of that language. If all the symbols used in the sentences are interpreted as objects of the problem space, the sentences can be called facts. (The use of the term, "facts" is my own terminology, but appears quite consistent.) A collection of facts makes up a model.

The above is an attempt to sketch the kinds of difficulties one must go through to rigorously transform a discussion about meaningless n-tuples to a discussion about meaningful facts forming a model of a problem space. We are not interested in such rigor, but in the notion that an n-tuple which has satisfactory form and interpretation can be asserted to have meaning. Furthermore, other meaningful facts (n-tuples) can be stated which place a structure on the problem space, by describing relations between relations and properties directly relevant to the problem space. Ultimately this structure can be used by deduction procedures to produce new facts about the problem space from the facts originally asserted.

This example presents a set of deduction procedures written in the STRAN language. Each procedure is based on a single primitive

relation which has proved useful in defining the characteristics of other relations. The primitive relations are described in some detail in the chapter on modeling. The deduction procedures presented here do not form a complete set. That is, they do not produce all possible true facts from a given set of facts. However, they can produce quite complicated deductions within the limited realm of their ability. The example shows several passes by the deduction procedures over a model of human relationships and some specific facts about humans and their relatives. The most complicated deduction accomplished is the fact that two persons are cousins.

The information stored in the associative memory when the deduction process begins, are the facts which were entered in examples 5 and 6 of this appendix.

BEGIN READING RULES.

```
INPUT...#  DEDUCTION RULES. CALLED BY DEDUCE.
INPUT...(DEDUCE(CHAIN,INVERSE,INTER,UNION,SUBSET,LEHALF,RIHALF,LINT,RINT,MINISET))
INPUT...(CHAIN(&F1|'CHAIN OF&AND'+$+$+$|=*OUT|'EXECUTING RULE OF CHAIN.'|)END,RCH1)
INPUT...(RCH1(&A1|1+2+3|)END,RCH2)
INPUT...(RCH2(&F2|2+$+$|)RCH1,RCH3)
INPUT...(RCH3(&A2|4+5|)RCH1,RCH4)
INPUT...(RCH4(&F3|3+5+$|)RCH3,RCH5)
INPUT...(RCH5(&A3|6|=&S|1+4+6|*OUT|' ('+1+','+4+','+6+')'|)RCH3,RCH5)
INPUT...(INVERSE(&F1|'INVERSE OF'+$+$|=*OUT|'EXECUTING RULE OF INVERSE.'|)END,RIV1)
INPUT...(RIV1(&A1|3+4|)END,RIV2)
INPUT...(RIV2(&F2|3+$+$|)RIV1,RIV3)
INPUT...(RIV3(&A2|1+2|=&S|4+2+1|*OUT|' ('+4+','+2+','+1+')'|)RIV1,RIV3)
INPUT...(INTER(&F1|'INTERSECTION OF&AND'+$+$+$|=*OUT|'EXECUTING INTER.'|)END,INT1)
INPUT...(INT1(&A1|1+2+3|)END,INT2)
INPUT...(INT2(=&S|'SUBSET OF'+1+2|*OUT|' (SUBSET OF,'+1+','+2+')'|)INT3)
INPUT...(INT3(=&S|'SUBSET OF'+1+3|*OUT|' (SUBSET OF,'+1+','+3+')'|)INT1)
INPUT...(UNION(&F1|'UNION OF&AND'+$+$+$|=*OUT|'EXECUTING RULE OF UNION.'|)END,UN1)
INPUT...(UN1(&A1|1+2+3|)END,UN2)
INPUT...(UN2(=&S|'SUBSET OF'+2+1|*OUT|' (SUBSET OF,'+2+','+1+')'|)UN3)
INPUT...(UN3(=&S|'SUBSET OF'+3+1|*OUT|' (SUBSET OF,'+3+','+1+')'|)UN1)
INPUT...(SUBSET(&F1|'SUBSET OF'+$+$|=*OUT|'EXECUTING RULE OF SUBSET.'|)END,SUB1)
INPUT...(SUB1(&A1|1+2|)END,SUB2)
INPUT...(SUB2(&F2|1+$|)SUB4,SUB3)
INPUT...(SUB3(&A2|3|=&S|2+3|*OUT|' ('+2+','+3+')'|)SUB4,SUB3)
INPUT...(SUB4(&F2|1+$+$|)SUB1,SUB5)
INPUT...(SUB5(&A2|3+4|=&S|2+3+4|*OUT|' ('+2+','+3+','+4+')'|)SUB1,SUB5)
INPUT...(LEHALF(&F1|'LEFT HALF OF'+$+$|=*OUT|'EXECUTING LEFT HALF.'|)END,LEH1)
INPUT...(LEH1(&A1|1+2|)END,LEH2)
INPUT...(LEH2(&F2|2+$+$|)LEH1,LEH3)
INPUT...(LEH3(&A2|3+4|=&S|1+3|*OUT|' ('+1+','+3+')'|)LEH1,LEH3)
```

```
INPUT...(RIHALF(&F1|'RIGHT HALF OF'+$+$|=*OUT|'EXECUTING RIGHT HALF.'|)END,RIH1)
INPUT...(RIH1(&A1|1+2|)END,RIH2)
INPUT...(RIH2(&F2|2+$+$|)RIH1,RIH3)
INPUT...(RIH3(&A2|3+4|=&S|1+4|*OUT|' ('+1+','+4+')'|)RIH1,RIH3)
INPUT...(LINT(&F1|'LEFT INTERSECTION OF&AND'+$+$+$|=*OUT|'EXECUTING LINT.'|)END,LIN1)
INPUT...(LIN1(&A1|1+2+3|)END,LIN2)
INPUT...(LIN2(&F2|1+$+$|)LIN5,LIN3)
INPUT...(LIN3(&A2|4+5|=&S|3+4|*OUT|' ('+3+','+4+')'|)LIN5,LIN4)
INPUT...(LIN4(=&S|2+4+5|*OUT|' ('+2+','+4+','+5+')'|)LIN3)
INPUT...(LIN5(&F2|2+$+$|)LIN1,LIN6)
INPUT...(LIN6(&A2|4+5|)LIN1,LIN7)
INPUT...(LIN7(&F3|3+4|=&S|1+4+5|*OUT|' ('+1+','+4+','+5+')'|)LIN6)
INPUT...(RINT(&F1|'RIGHT INTERSECTION OF&AND'+$+$+$|=*OUT|'EXECUTING RINT.'|)END,RIN1)
INPUT...(RIN1(&A1|1+2+3|)END,RIN2)
INPUT...(RIN2(&F2|1+$+$|)RIN5,RIN3)
INPUT...(RIN3(&A2|4+5|=&S|3+5|*OUT|' ('+3+','+5+')'|)RIN5,RIN4)
INPUT...(RIN4(=&S|2+4+5|*OUT|' ('+2+','+4+','+5+')'|)RIN3)
INPUT...(RIN5(&F2|2+$+$|)RIN1,RIN6)
INPUT...(RIN6(&A2|4+5|)RIN1,RIN7)
INPUT...(RIN7(&F3|3+5|=&S|1+4+5|*OUT|' ('+1+','+4+','+5+')'|)RIN6)
INPUT...(MINISET(&F1|'MINISET OF&AND'+$+$+$|=*OUT|'EXECUTING MINISET.'|)END,MIN1)
INPUT...(MIN1(&A1|2+3+4|=&S|2+4|*OUT|' ('+2+','+4+')'|)END,MIN2)
INPUT...(MIN2(=&S|'SUBSET OF'+3+2|*OUT|'(SUBSET OF,'+3+','+2+')'|)MIN1)
INPUT...(DEDUCE)
```

BEGIN INTERPRETING RULES.

```
EXECUTING RULE OF CHAIN.
 (BROTHER IN LAW OF,*BARTON,*MADELAINE)
 (FATHER IN LAW OF,*MARVIN,*MADELAINE)
 (COMES FROM,*MADELAINE,*FRANCE)
EXECUTING RULE OF INVERSE.
 (MARRIED TO,*ALICE,*MARVIN)
 (MARRIED TO,*MERLA,*BARTON)
 (MARRIED TO,*MADELAINE,*NORMAN)
EXECUTING INTER.
EXECUTING RULE OF UNION.
EXECUTING RULE OF SUBSET.
 (UNMARRIED,*ERIC)
 (OFFSPRING,*ERIC)
 (PERSON,*ERIC)
 (MALE,*ERIC)
 (MALE PERSON,*ERIC)
 (CHILD,*ERIC)
 (PERSON,*CHRISTOPHER)
 (UNMARRIED,*CHRISTOPHER)
 (OFFSPRING,*CHRISTOPHER)
EXECUTING LEFT HALF.
 (FATHER,*NORMAN)
 (FATHER,*MARVIN)
 (BROTHER IN LAW,*BARTON)
 (FATHER IN LAW,*MARVIN)
 (MARRIED,*MARVIN)
 (MARRIED,*MADELAINE)
 (MARRIED,*MERLA)
 (MARRIED,*ALICE)
 (MARRIED,*BARTON)
```

```
 (MARRIED,*NORMAN)
 (BROTHER,*BARTON)
 (DAUGHTER,*PAT)
 (SON,*KEVIN)
 (SON,*CHRISTOPHER)
EXECUTING RIGHT HALF.
 (OFFSPRING,*NORMAN)
EXECUTING LINT.
 (MALE,*BARTON)
 (SIBLING OF,*BARTON,*NORMAN)
 (MAN,*NORMAN)
 (PARENT OF,*NORMAN,*ERIC)
 (MAN,*MARVIN)
 (PARENT OF,*MARVIN,*NORMAN)
 (FEMALE,*PAT)
 (OFFSPRING OF,*PAT,*ALICE)
 (MALE,*KEVIN)
 (OFFSPRING OF,*KEVIN,*BARTON)
 (OFFSPRING OF,*CHRISTOPHER,*NORMAN)
 (HUSBAND OF,*MARVIN,*ALICE)
 (HUSBAND OF,*NORMAN,*MADELAINE)
```

```
BEGIN INTERPRETING RULES.

        EXECUTING RULE OF CHAIN.
         (NEPHEW OF,*KEVIN,*NORMAN)
         (UNCLE OF,*BARTON,*ERIC)
         (GRANDFATHER OF,*MARVIN,*ERIC)
         (GRANDPARENT OF,*MARVIN,*ERIC)
        EXECUTING RULE OF INVERSE.
         (PARENT OF,*ALICE,*PAT)
         (PARENT OF,*NORMAN,*CHRISTOPHER)
         (PARENT OF,*BARTON,*KEVIN)
         (WIFE OF,*ALICE,*MARVIN)
         (WIFE OF,*MADELAINE,*NORMAN)
         (SIBLING OF,*NORMAN,*BARTON)
         (OFFSPRING OF,*ERIC,*NORMAN)
         (OFFSPRING OF,*NORMAN,*MARVIN)
        EXECUTING INTER.
        EXECUTING RULE OF UNION.
        EXECUTING RULE OF SUBSET.
         (MALE PERSON,*NORMAN)
         (MALE PERSON,*MARVIN)
         (MALE,*NORMAN)
         (MALE,*MARVIN)
         (OFFSPRING,*MARVIN)
         (ADULT,*NORMAN)
         (ADULT,*MARVIN)
         (PARENT,*NORMAN)
         (PARENT,*MARVIN)
         (OFFSPRING,*PAT)
         (OFFSPRING,*KEVIN)
         (SIBLING,*BARTON)
         (PERSON,*MARVIN)
```

```
 (PERSON,*NORMAN)
 (OFFSPRING,*BARTON)
 (OFFSPRING,*ALICE)
 (OFFSPRING,*MERLA)
 (OFFSPRING,*MADELAINE)
 (ADULT,*BARTON)
 (ADULT,*ALICE)
 (ADULT,*MERLA)
 (ADULT,*MADELAINE)
 (PERSON,*BARTON)
 (PERSON,*ALICE)
 (PERSON,*MERLA)
 (PERSON,*MADELAINE)
EXECUTING LEFT HALF.
 (GRANDFATHER,*MARVIN)
 (SIBLING,*NORMAN)
 (WIFE,*ALICE)
 (WIFE,*MADELAINE)
 (HUSBAND,*MARVIN)
 (HUSBAND,*NORMAN)
 (UNCLE,*BARTON)
 (PARENT,*BARTON)
 (PARENT,*ALICE)
EXECUTING RIGHT HALF.
EXECUTING LINT.
 (CHILD OF,*ERIC,*NORMAN)
 (CHILD OF,*CHRISTOPHER,*NORMAN)
 (BROTHER OF,*NORMAN,*BARTON)
 (FATHER OF,*NORMAN,*CHRISTOPHER)
 (SON OF,*NORMAN,*MARVIN)
 (SON OF,*ERIC,*NORMAN)
 (WOMAN,*ALICE)
 (WOMAN,*MADELAINE)
```

```
BEGIN INTERPRETING RULES.

        EXECUTING RULE OF CHAIN.
         (DAUGHTER IN LAW OF,*MADELAINE,*MARVIN)
         (SISTER IN LAW OF,*MADELAINE,*BARTON)
         (BROTHER IN LAW OF,*NORMAN,*MERLA)
         (COUSIN OF,*KEVIN,*ERIC)
         (COUSIN OF,*KEVIN,*CHRISTOPHER)
         (NEPHEW OF,*ERIC,*BARTON)
         (NEPHEW OF,*CHRISTOPHER,*BARTON)
         (UNCLE OF,*BARTON,*CHRISTOPHER)
         (UNCLE OF,*NORMAN,*KEVIN)
         (GRANDCHILD OF,*ERIC,*MARVIN)
         (GRANDCHILD OF,*CHRISTOPHER,*MARVIN)
         (GRANDSON OF,*ERIC,*MARVIN)
         (GRANDSON OF,*CHRISTOPHER,*MARVIN)
         (GRANDFATHER OF,*MARVIN,*CHRISTOPHER)
         (GRANDPARENT OF,*MARVIN,*CHRISTOPHER)
        EXECUTING RULE OF INVERSE.
         (COUSIN OF,*ERIC,*KEVIN)
         (COUSIN OF,*CHRISTOPHER,*KEVIN)
        EXECUTING INTER.
        EXECUTING RULE OF UNION.
        EXECUTING RULE OF SUBSET.
         (FEMALE PERSON,*ALICE)
         (FEMALE PERSON,*MADELAINE)
         (FEMALE,*ALICE)
         (FEMALE,*MADELAINE)
         (GRANDPARENT,*MARVIN)
        EXECUTING LEFT HALF.
         (DAUGHTER IN LAW,*MADELAINE)
         (SISTER IN LAW,*MADELAINE)
```

```
 (BROTHER IN LAW,*NORMAN)
 (GRANDSON,*ERIC)
 (GRANDSON,*CHRISTOPHER)
 (BROTHER,*NORMAN)
 (COUSIN,*KEVIN)
 (COUSIN,*CHRISTOPHER)
 (COUSIN,*ERIC)
 (UNCLE,*NORMAN)
 (SON,*ERIC)
 (SON,*NORMAN)
EXECUTING RIGHT HALF.
EXECUTING 1 INT.
 (MOTHER OF,*ALICE,*PAT)
```

B.8 Example 8

In example 6 we showed how simple English sentences can be parsed into sentences of STL. That example was carried out first, because parsing is usually considered to be more difficult than sentence generation. However, it is also important to be able to generate English sentences from STL n-tuples. In this example a rule set is used to take the n-tuples produced by example 6 back to English again. Thus, the processes of parsing and generation are easily reversible for these rule sets. This is not always the case in linguistic methods.

A number of n-tuples concerning the syntactic categories of various syntactic terms used by the parser are introduced first. The final sentence of that collection is "*NEUTER IS AN ADJECTIVE." All sentences following that, result from n-tuples from the parsing in example 6. The character * is used to indicate that the following letter is capitalized, since the line printer does not have lower case characters. All sentences beginning with * have been generated, by the rule set, from a preceding input n-tuple.

```
STRAN INTERPRETER ON MAY 22,1969 AT 05:50:00.580 PAGE 18


BEGIN READING RULES.


INPUT...# THE FOLLOWING CARDS, S4 TO S2, SUBSTITUTE FOR RULES WHICH ARE NORMALLY PART OF
INPUT...# OF ASSERT.  THE NORMAL RULES PRINT OUT A STORED N-TUPLE IN S.T.L.
INPUT...# THESE RULES PRINT OUT THE STORED N-TUPLE IN ENGLISH.
INPUT...(S4(OUT|$+',',+$+',',+$+',',+$|=O1|'.'|&S|1+3+5+7|RLTN|1|SBJ|3|O1|5|O2|7|)S3,W4)
INPUT...(S3(OUT|$+',',+$+',',+$|=SBJ|'.'|&S|1+3+5|RLTN|1|SBJ|3|O1|5|)S2,W3)
INPUT...(S2(OUT|$+',',+$|=SBJ|'.'|&S|1+3|PHRASE|1|SBJ|3|)S1,W2)
INPUT...
INPUT...# THE FOLLOWING RULES GENERATE ENGLISH SENTENCES FROM S.T.L. N-TUPLES.
INPUT...(W4(O1|'.'|)W4A,END)
INPUT...(W3(SBJ|'.'|)W3A,END)
INPUT...(W2(SBJ|'.'|)WA,END)
INPUT...(WA(PHRASE|$|&F9|'NOUN'+1|=TMP|'ADET,WD))'|)WD,TMP)
INPUT...(WB(SBJ|'*'+$|PHRASE|$|=*OUTPUT|1+2+3+'.'|)WC,END)
INPUT...(WC(SBJ|$|PHRASE|$|=*OUTPUT|'*'+1+2+'.'|)END)
INPUT...(WD(PHRASE|$|=PHRASE|' IS '+1|)WB)
INPUT...(W3A(RLTN|'SUBSET OF'|C1|$|=PHRASE|2|MARK|'*ANY'|)W3B,W3A1)
INPUT...(W3B(RLTN|'DISJOINT FROM'|O1|$|=PHRASE|2|MARK|'*NO'|)W3P,W3A1)
INPUT...(W3A1(MARK|$|SBJ|$|&F9|'NOUN'+2|=SBJ|1+' '+2|)W3A2,WA)
INPUT...(W3A2(=SBJ|1+'THING '+2|)WA)
INPUT...(W3P(RLTN|$|O1|$|=PHRASE|1+' '+2|)W3C)
INPUT...(W3C(&F9|'FUNCTION'+1|=TMP|'TDET,WD))'|)W3D,TMP)
INPUT...(W3D(&F9|'NOUN ROOT'+1|=TMP|'ADET,WD))'|)W3E,TMP)
INPUT...(W3E(&F9|'VERB ROOT'+1|PHRASE|$|=PHRASE|' '+1|)WD,WB)
INPUT...(ADET(PHRASE|$1+$|&F9|'VOWEL'+1|=PHRASE|'AN '+1+2|)BDET,END)
INPUT...(BDET(=PHRASE|'A '+1+2|)END)
INPUT...(TDET(PHRASE|$|=PHRASE|'THE '+1|)END)
INPUT...(W4A(RLTN|$+'&'+$|O1|$|O2|$|=PHRASE|1+' '+4+' '+3+' '+5|)W4A1,W4B)
INPUT...(W4A1(RLTN|$|=*OUTPUT|'('+1+','+3+4+5+6+7+')'|)END)
INPUT...(W4B(RLTN|$|)W3C)
INPUT...(ASSERT)
```

BEGIN INTERPRETING RULES.

```
INPUT...  THE FOLLOWING ARE FACTS ABOUT THE SYNTACTIC CATEGORIES OF WORDS AND PHRASES
INPUT...  USED BY THE PARSER TO INDICATE SYNTACTIC INFORMATION.
INPUT...(NOUN,NOUN)(NOUN,ADJECTIVE)(NOUN,NOUN ROOT)(NOUN,VERB ROOT)
        *NOUN IS A NOUN.

        *ADJECTIVE IS A NOUN.

        *NOUN ROOT IS A NOUN.

        *VERB ROOT IS A NOUN.

INPUT...(NOUN,VOWEL)(VOWEL,A)(VOWEL,E)(VOWEL,I)(VOWEL,O)(VOWEL,U)
        *VOWEL IS A NOUN.

        *A IS A VOWEL.

        *E IS A VOWEL.

        *I IS A VOWEL.

        *O IS A VOWEL.

        *U IS A VOWEL.

INPUT...(NOUN,FUNCTION)(NOUN,PROPER NAME)(NOUN,MALE)(NOUN,FEMALE)
        *FUNCTION IS A NOUN.

        *PROPER NAME IS A NOUN.

        *MALE IS A NOUN.
```

```
        *FEMALE IS A NOUN.

INPUT...(NOUN,VERB PHRASE)(ADJECTIVE,UNIQUE)(NOUN,ADJECTIVE PHRASE)
        *VERB PHRASE IS A NOUN.

        *UNIQUE IS AN ADJECTIVE.

        *ADJECTIVE PHRASE IS A NOUN.

INPUT...(NOUN ROOT,SUBSET OF)(ADJECTIVE,NEUTER)
        *SUBSET OF IS A NOUN ROOT.

        *NEUTER IS AN ADJECTIVE.

INPUT...  THE FOLLOWING STATEMENTS ARE THE RESULTS OF EARLIER PARSINGS.
INPUT...(PROPER NAME,*ERIC)
        *ERIC IS A PROPER NAME.

INPUT...(FUNCTION,FATHER OF)
        *FATHER OF IS A FUNCTION.

INPUT...(NOUN ROOT,FATHER OF)
        *FATHER OF IS A NOUN ROOT.

INPUT...(PROPER NAME,*NORMAN)
        *NORMAN IS A PROPER NAME.

INPUT...(FATHER OF,*NORMAN,*ERIC)
        *NORMAN IS THE FATHER OF *ERIC.

INPUT...(PROPER NAME,*MARVIN)
        *MARVIN IS A PROPER NAME.
```

```
INPUT...(FATHER OF,*MARVIN,*NORMAN)
        *MARVIN IS THE FATHER OF *NORMAN.

INPUT...(PROPER NAME,*ALICE)
        *ALICE IS A PROPER NAME.

INPUT...(NOUN ROOT,DAUGHTER OF)
        *DAUGHTER OF IS A NOUN ROOT.

INPUT...(PROPER NAME,*PAT)
        *PAT IS A PROPER NAME.

INPUT...(DAUGHTER OF,*PAT,*ALICE)
        *PAT IS A DAUGHTER OF *ALICE.

INPUT...(ADJECTIVE,LITTLE)
        *LITTLE IS AN ADJECTIVE.

INPUT...(LITTLE,*ERIC)
        *ERIC IS LITTLE.

INPUT...(NOUN,BOY)
        *BOY IS A NOUN.

INPUT...(BOY,*ERIC)
        *ERIC IS A BOY.

INPUT...(NOUN ROOT,BROTHER OF)
        *BROTHER OF IS A NOUN ROOT.

INPUT...(PROPER NAME,*BARTON)
        *BARTON IS A PROPER NAME.
```

```
INPUT...(BROTHER OF,*BARTON,*NORMAN)
        *BARTON IS A BROTHER OF *NORMAN.

INPUT...(MARRIED TO,*MARVIN,*ALICE)
        *MARVIN IS MARRIED TO *ALICE.

INPUT...(PROPER NAME,*MADELAINE)
        *MADELAINE IS A PROPER NAME.

INPUT...(MARRIED TO,*NORMAN,*MADELAINE)
        *NORMAN IS MARRIED TO *MADELAINE.

INPUT...(PROPER NAME,*MERLA)
        *MERLA IS A PROPER NAME.

INPUT...(MARRIED TO,*BARTON,*MERLA)
        *BARTON IS MARRIED TO *MERLA.

INPUT...(NOUN ROOT,SON OF)
        *SON OF IS A NOUN ROOT.

INPUT...(PROPER NAME,*KEVIN)
        *KEVIN IS A PROPER NAME.

INPUT...(SON OF,*KEVIN,*BARTON)
        *KEVIN IS A SON OF *BARTON.

INPUT...(PROPER NAME,*CHRISTOPHER)
        *CHRISTOPHER IS A PROPER NAME.

INPUT...(SON OF,*CHRISTOPHER,*NORMAN)
        *CHRISTOPHER IS A SON OF *NORMAN.
```

```
INPUT...(MALE,*CHRISTOPHER)
        *CHRISTOPHER IS A MALE.

INPUT...(NOUN,CHILD)
        *CHILD IS A NOUN.

INPUT...(CHILD,*CHRISTOPHER)
        *CHRISTOPHER IS A CHILD.

INPUT...(PROPER NAME,*FRANCE)
        *FRANCE IS A PROPER NAME.

INPUT...(PROPER NAME,*MANDRES)
        *MANDRES IS A PROPER NAME.

INPUT...(IN,*MANDRES,*FRANCE)
        *MANDRES IS IN *FRANCE.

INPUT...(VERB ROOT,COMES FROM)
        *COMES FROM IS A VERB ROOT.

INPUT...(COMES FROM,*MADELAINE,*MANDRES)
        *MADELAINE COMES FROM *MANDRES.

INPUT...))))))
```

APPENDIX C

THE ASSOCIATIVE MEMORY, HASH CODING AND FACT RETRIEVAL

C.1 Hash Coding

Hash coding is the process of producing an integer number when given an arbitrary character string, by some fixed manipulation of that character string. A good hash coder, when given a large set of different character strings, will produce numbers which are evenly distributed over a given interval. To produce a hash coder, is to produce a particular many-one map from sets of character strings, to a given set of integers.

The numbers which are produced by the hash-coder may be used as indices to a table, in order to "look-up" desired information associated with the character string. Since more than one character string can map into a given integer, a difficulty which is usually called overlap, information must be stored at each table location indicating exactly what character string caused data to be stored there.

A particularly lucid way to talk about hash-coding is to describe any character string and its associated information as forming a name-data pair. A notation for this is: (N,D). If we define H_{mn} to be the m-th many-one map from character strings to the integers from 1 to n, then the fact that overlap occurs for two names

N and M under H_{mn} means that $H_{mn}(N) = H_{mn}(M)$. Another difficulty which can occur in hash coding is that more than one data item may be associated with a given name (eg. $(N,D_1),(N,D_2)$). This is usually called multiplicity. Thus, for any given table location, more than one name may have caused information to be stored there (overlap) and each of those names may have more than one associated data item (multiplicity). Multiplicity is a function only of the characteristics of the data, and is unaffected by changing the hash-coder. Overlap is related to ineffectiveness of the hash-coder and the ratio of the number of different names under which storage has occurred to n (the size of the set of integers). The fact that a particular set of names produces a great deal of overlap is not conclusive evidence of a bad hash-coder, since given a hash-coder one can always produce a set of names which map onto the same integer. However, if the set of names is large and arbitrarily chosen, and the hash-coder is effective, such an occurrence is an extremely unlikely event.

In the material that follows we will examine two general methods of handling overlap and two methods for multiplicity. Next we will describe how those methods must be adapted when the problem is one of storing and retrieving n-tuples of character strings instead of a single string. Finally, we will describe actual hash-coding methods used by the associative memory and the STRAN language interpreter.

The two general methods for handling overlap are:

(1) Chaining - Use of a linked list of cells beginning at the table location indicated by the integer from the hash-coder.

(2) Generation of sequences of indices - If the initial integer references a table location which is full, the integer is passed to a function which produces a new integer, and so on, until an empty location is found.

The second process above is often called "rehashing", but that term fails to indicate that the function in question may actually be highly non-random, perhaps simply adding one to the previous integer. The first method is sometimes referred to as the use of "stretchable buckets", indicating that each table location has been given the capability of holding an arbitrary amount of information.

The two general methods for handling multiplicity are nearly identical to the two methods we have just described for handling overlap. They are:

(1) Chaining - The data items for a particular name are kept on a linked list (chain) with the name at its head. This multiplicity list is a sublist of the overlap list.

(2) Generation of index sequences. - The data items are stored at a sequence of table locations whose indices are generated algorithmically. Great care must be exercised if the index sequence method is also being used for overlap, for the sequences of indices will interact. In that case each data item in a multiplicity index sequence must be marked with its position in the sequence, to prevent overlapping of other data items from interferring with its successful retrieval. This statement will be clarified in the examples to follow.

C.2 Overlap Techniques

In the previous section we described overlap, (the hash coder produces the same integer for two different names) and mentioned two techniques for handling it. Those two techniques are chaining and generation of index sequences. In this section we will describe those techniques in greater detail. For simplicity, we will assume that multiplicity does not occur, i.e. only one data item will be associated with each name. Multiplicity techniques will be clarified in the next section.

In the chaining technique of handling overlap, an integer is generated from the name portion of the name-data pair and that integer is used as an index to a table. That table location will be

the first cell of a linked list of name-data pairs which have hash-coded to the same integer. The list may be empty, indicating nothing has been stored at this location. More than one name-data pair on the list indicates that overlap has occurred.

In retrieval of name-data pairs, the overlap list must be searched for the particular name being sought. If the name has been stored it will be found, on the average, half way down the overlap list. If the name has not been stored, the complete overlap list must be fruitlessly searched to prove the absence of the desired name.

The chaining technique has the advantage that the list of pairs contains only names which have hash coded to the same integer. This is not true of the index sequence technique. This efficiency is achieved at the cost of the extra storage required for address pointers to the next member of the linked list.

In the index sequence method of handling overlap, a hash-coder is constructed which is capable of generating a sequence of integers in the interval from 1 to n when presented with a given name (character string). Thus, presented with N, the hash-coder generates first the integer I_1. When requested to give the next integer corresponding to N, it generates I_2, then I_3, etc. The sequence I_1, I_2, I_3,... is repeatable i.e., for a given name N the sequence will always be the same.

Under the index sequence method of handling overlap (N,D) is stored at position I_1 of the table, unless that spot is already full,

in which case (N,D) is stored at I_2, unless that spot is full, etc. Since the interval on which the integers are produced is bounded, the index sequence must ultimately contain repetitions. This means that when the table becomes nearly full, the storage algorithm may go into an infinite loop looking for a free location. This particular problem cannot occur in the chained list method of handling overlap where storing of (N,D) pairs ceases when no more space is available for extension of the linked lists.

The retrieval of information using the index sequence method requires the probing of locations indexed by I_1, I_2, I_3,... until the desired name or an empty cell is found.

An important concept with reference to overlap is the fact that whenever overlap becomes large it can always be reduced by storing the same information over a larger set of integers. Changing to a larger set of integers usually requires destruction of previously stored information and starting from scratch again. For large amounts of information this may become an expensive proposition. Thus, for large and continuously growing sets of information, some thought must be given to the initial size of the set of integers versus the probable difficulty of restoring the information over a larger set at some later time.

C.3 Multiplicity Techniques

Multiplicity occurs when several data items are associated with the same name. As was mentioned in the initial section of this chapter, this problem can be handled by chaining or index sequence methods.

The chaining method works in obvious fashion, linking together data items on a list headed by the name with which they associate. It is important, of course, that the method of handling multiplicity be distinguishable from that used to handle overlap. Thus, if chaining is used for both overlap and multiplicity, the multiplicity lists must be sublists on the overlap chains.

In the index sequence method of handling multiplicity, a preprocess is carried out on the name before hash coding, to produce a sequence of unique versions of that name for each data item to be stored with it. Each of the sequence of preprocessed versions of the name will be represented by subscripting in the example to follow. Thus, the first data item will be stored under the name N_1, the second under N_2, etc. When an (N,D) pair is to be stored, attempts must be made for N_1, N_2,... until it can be stored as (N_j,D) for some j. An increase in efficiency can be achieved by storing the highest value of the subscript j for which data is stored under the name N_1 so that the first unsuccessful attempt to store under N_1 gives the information that N_{j+1} is the form of the name under which to store the data item.

As was stated earlier, care must be exercised to assure that multiplicity and overlap are distinguishable. This is because in searching for a name N_i, the i-th member of a multiplicity list, where the index sequence method is being used for overlap, another member of the same multiplicity sequence N_k ($k \neq i$) may appear in the overlap sequence and be accepted as representing the i-th data item. This can only occur if we are unable to distinguish between the i-th and k-th member of the multiplicity index sequence.

C.4 Hash Coded Storage and Retrieval of Ordered N-tuples of Symbols

Now that we have examined hash coding of single strings, we are ready to examine the problem of hash-coding n-tuples of strings. In particular, we will describe an encoding of n-tuples of character strings as name-data pairs which allows retrieval of an n-tuple when one or more of its symbols are known.

Up to now we have been able to talk freely about character strings and symbols interchangeably, but now we must make some notational definitions in order to distinguish an n-tuple of symbols from a symbol. A symbol is arbitrarily defined to be any character string which contains no parentheses or commas. An n-tuple is a collection of n symbols, beginning with left parenthesis, ending with right parenthesis, and separated internally by commas. As it happens, no need has been found for dealing with n-tuples where n is greater than four, so no discussion of larger n-tuples will occur here. However, in case a

need is ever found, the only difficulty will be in allocating enough storage to handle bigger n-tuples.

For each n-tuple (here to be called a "closed expression") we generate 2^n-2 open expressions, each containing at least one symbol from the original n-tuple. The open expressions will serve as names, while the closed expression will be the data. For example, consider the n-tuple:

(TALL,BOB)

For this 2-tuple there are $2^2-2 = 2$ open expressions, generated by replacing the symbols of the 2-tuple by the symbol $ (which is understood to stand for the empty symbol). The open expressions generated are:

($,BOB)

(TALL,$)

Since we will also want to be able to retrieve any n-tuple by its closed form as well as the open forms, the closed form must also be used as a name. Thus, for any given n-tuple there will be 2^n-1 names (2^n-2 open forms and one closed form) with which that n-tuple must be stored as data.

Other workers in the field of hash coded retrieval of n-tuples (among them Feldman, 1965) have not deemed it necessary to store under all the possible open forms of the n-tuple, and have picked a particular set of forms which they felt would be most frequently used. Such

a decision is difficult to make when one is not certain exactly what uses will be made of the storage mechanism, so in the associative memory for COMS all the open forms plus the closed form are used as names.

For the 3-tuple (FATHER OF, MARVIN, NORMAN) this means that the hash coded associative storage must receive the following seven name-data pairs.

	NAME	DATA
(1)	(FATHER OF, MARVIN, NORMAN)	(FATHER OF, MARVIN, NORMAN)
(2)	($, MARVIN, NORMAN)	" " "
(3)	(FATHER OF, $, NORMAN)	" " "
(4)	(FATHER OF, MARVIN, $)	" " "
(5)	($, $, NORMAN)	" " "
(6)	($, MARVIN, $)	" " "
(7)	(FATHER OF, $, $)	" " "

C.5 An Example

Under a contract held by Computer Corporation of America, the author implemented two hash-coded associative memories for n-tuples. The first, called the AM-1a, used index sequence techniques for handling multiplicity and overlap. The second, called the AM-2b, used chaining techniques for overlap and multiplicity. These mechanisms were implemented on the IBM 7094 in FORTRAN and FAP, and statistics concerning the operation of those programs are reported by Gammill, Marill and

Yates (1967). The hash-storage for n-tuples (associative memory) used by the STRAN interpreter is nearly identical to the AM-2b. In the example which follows we demonstrate how two 2-tuples would be stored in the AM-1a and AM-2b. The material concerning the AM-1a is included to show another way the associative n-tuple memory could have been implemented. It is also included because the rule-name and symbol dictionaries used by the interpreter (hash coding of a character string to produce an integer code number for internal use) are both carried out by index sequence techniques.

In the example we show the data structure resulting from the storage of two 2-tuples, (COUNTRY, AUSTRALIA) and (COUNTRY, MEXICO), by the AM-1a and AM-2b. Flow diagrams for these two processors are given in Gammill, Marill and Yates (1967) and the material for the example appeared in that report. The processes described take place internally and do not have to be understood by a user. It is assumed that the memory is empty at the start of the example.

First, corresponding to each expression to be stored, three name-data pairs are developed, yielding six in all. They are as follows:

NAME	DATA
(COUNTRY, AUSTRALIA)	(COUNTRY, AUSTRALIA)
($, AUSTRALIA)	(COUNTRY, AUSTRALIA)
(COUNTRY, $)	(COUNTRY, AUSTRALIA)
(COUNTRY, MEXICO)	(COUNTRY, MEXICO)
($, MEXICO)	(COUNTRY, MEXICO)
(COUNTRY, $)	(COUNTRY, MEXICO)

C.5.1 The AM-1a

Assume we are dealing with an AM-1a. The program uses the index sequence method for handling overlap. That is, it first obtains an integer H(name) corresponding to name. If location H(name) is occupied, it will generate a new integer by computing G(H(name)); if further integers are needed, it will obtain G(G(H(name))), etc. The function H takes an arbitrary character string into an integer. The function G takes an integer into an integer.

Multiplicity is also handled by the index sequence technique. In this approach, all names are considered to have subscripts. When the program tries to store a name-data pair and discovers that the given name has already been used with another data-item, it increments the subscript of the name by 1, thereby generating a new name. In effect, the subscripted names form an index sequence.

Assume that we have a hash-table with eight locations. We obtain the integer table index corresponding to the first name, (COUNTRY, AUSTRALIA), obtaining $H((\text{COUNTRY, AUSTRALIA})_1) = 7$. The first name-data pair is then stored at location 7.

To store the second name-data pair, we obtain $H((\$, \text{AUSTRALIA})_1) = 7$. Since location 7 has already been used by a different name (overlap), we cannot store at 7, but must generate another integer, obtaining $G(H((\$, \text{AUSTRALIA})_1)) = 4$. Since 4 is unoccupied, we stc the second name-data pair at 4.

To store the third pair, we obtain $H((\text{COUNTRY}, \$)_1) = 3$. Since 3 is unoccupied, the third pair is stored at 3.

To store the fourth pair, we obtain $H((\text{COUNTRY}, \text{MEXICO})_1) = 3$. However, 3 is already occupied with a pair having a different name, so we must obtain a new integer, getting $G(H((\text{COUNTRY}, \text{MEXICO})_1)) = 6$. Since 6 is unoccupied, the fourth pair is stored at 6.

To store the fifth pair we obtain the integer $H((\$, \text{MEXICO})_1) = 1$. Since 1 is unoccupied, the fifth pair is stored at 1.

To store the sixth pair, we obtain $H((\text{COUNTRY}, \$)_1) = 3$, as before. We find that 3 is already occupied by a name-data pair having the same name but different data (multiplicity). We increment the subscript and hash again, obtaining $H((\text{COUNTRY}, \$)_2) = 1$. However, 1 is already occupied by a name-data pair having a different name (overlap), so we generate the next integer $G(H((\text{COUNTRY}, \$)_2)) = 2$. Since 2 is empty, the sixth pair is stored at 2.

The final state of the memory is represented below:

LOCATION	NAME	DATA
1	($, MEXICO)$_1$	(COUNTRY, MEXICO)
2	(COUNTRY, $)$_2$	(COUNTRY, MEXICO)
3	(COUNTRY, $)$_1$	(COUNTRY, AUSTRALIA)
4	($, AUSTRALIA)$_1$	(COUNTRY, AUSTRALIA)
5	---	---
6	(COUNTRY, MEXICO)$_1$	(COUNTRY, MEXICO)
7	(COUNTRY, AUSTRALIA)$_1$	(COUNTRY, AUSTRALIA)
8	---	---

Normally one would not allow a hash-field to become so crowded, since one would be interested in holding overlap down to a minimum. This crowded example is given here to show index sequence methods at work.

C.5.2 The AM-2b

Assume that we are dealing with an AM-2b. The program will use the chaining (list) technique for handling both overlap and multiplicity. In the hash coding, we can dispense with the subscripts on names, and with the index sequence function G. The structure of the linked lists will now be shown. A name-data pair is represented as shown in Fig. C.1.

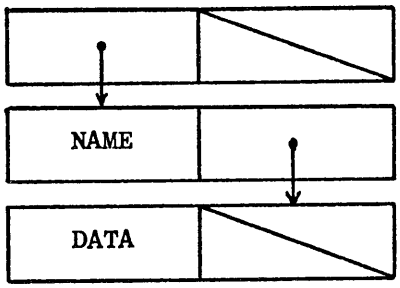


Fig. C.1 A single name-data pair represented in a linked list.

The first cell (some fixed unit of computer storage, frequently a word) is located in the hash table. If overlap occurs, the structure is extended "laterally" as shown in Fig. C.2. If multiplicity occurs, the list is extended vertically as shown in Fig. C.3.

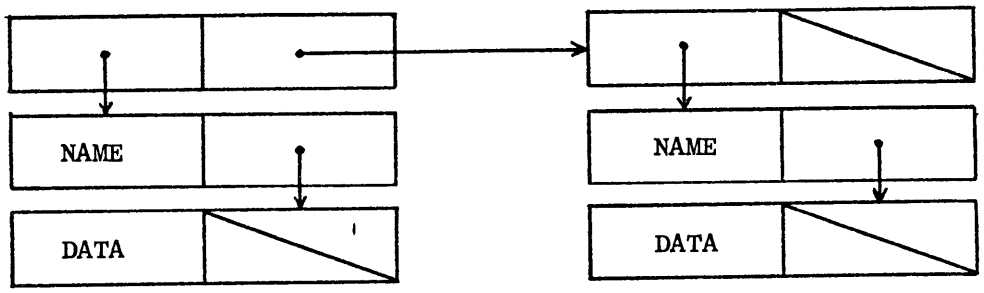


Fig. C.2 Two name-data pairs linked to form an overlap list.

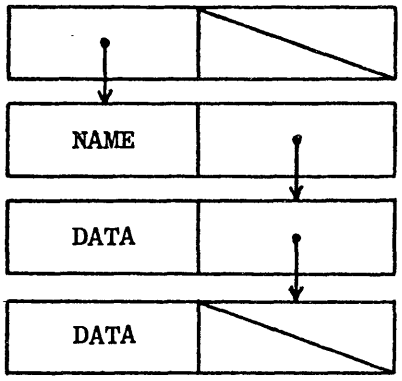


Fig. C.3 A name and two data items linked to form a multiplicity list.

For name-data pairs in which the name and the data are identical (as in the first and fourth pairs of our example), it is not necessary for the AM-2b to store both the name and the data so only the name is stored, as shown in Fig. C.4.

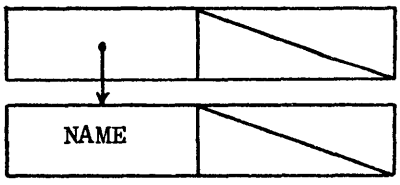


Fig. C.4 Storage of a closed form, with name and data identical.

For the sake of the example assume that:

H(COUNTRY, AUSTRALIA) = 8

H($, AUSTRALIA) = 8

H(COUNTRY, $) = 1

H(COUNTRY, $) = 1

H(COUNTRY, MEXICO) = 1

H($, MEXICO) = 5

H(COUNTRY, $) = 1

In practice, such a high degree of overlap would be very rare. Thus, if the two expressions (COUNTRY, AUSTRALIA) and (COUNTRY, MEXICO) are given to an AM-2b for storage, the data structure shown in Fig. C.5 will result.

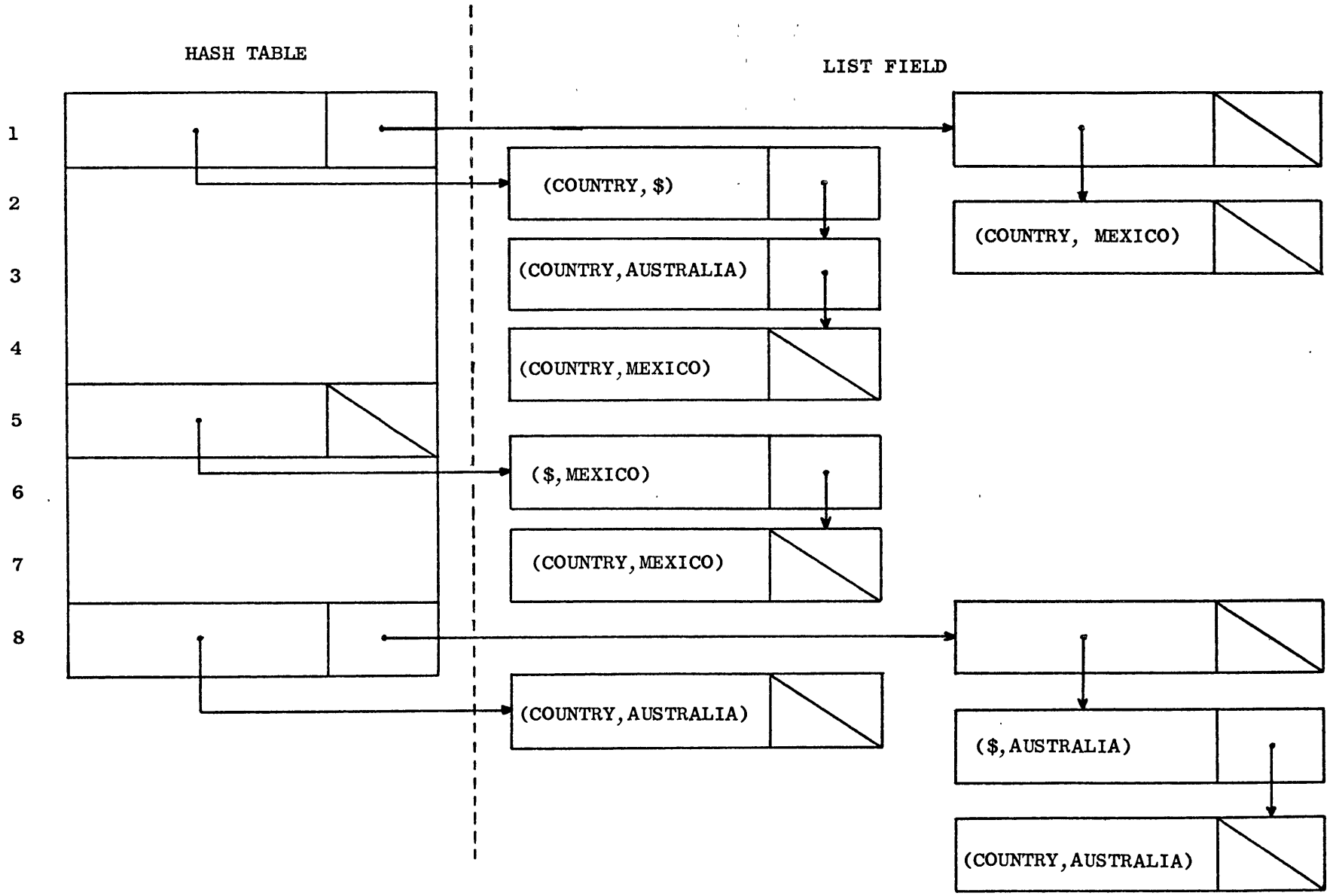


Fig. C.5 Two 2-tuples stored in an AM-2b.

C.6 Some Theoretical Characteristics of Hash-Coding

The process of finding objects in any kind of memory mechanism can be broken up into parts. One of the ways it can be described is flow diagrammed in Fig. C.6.

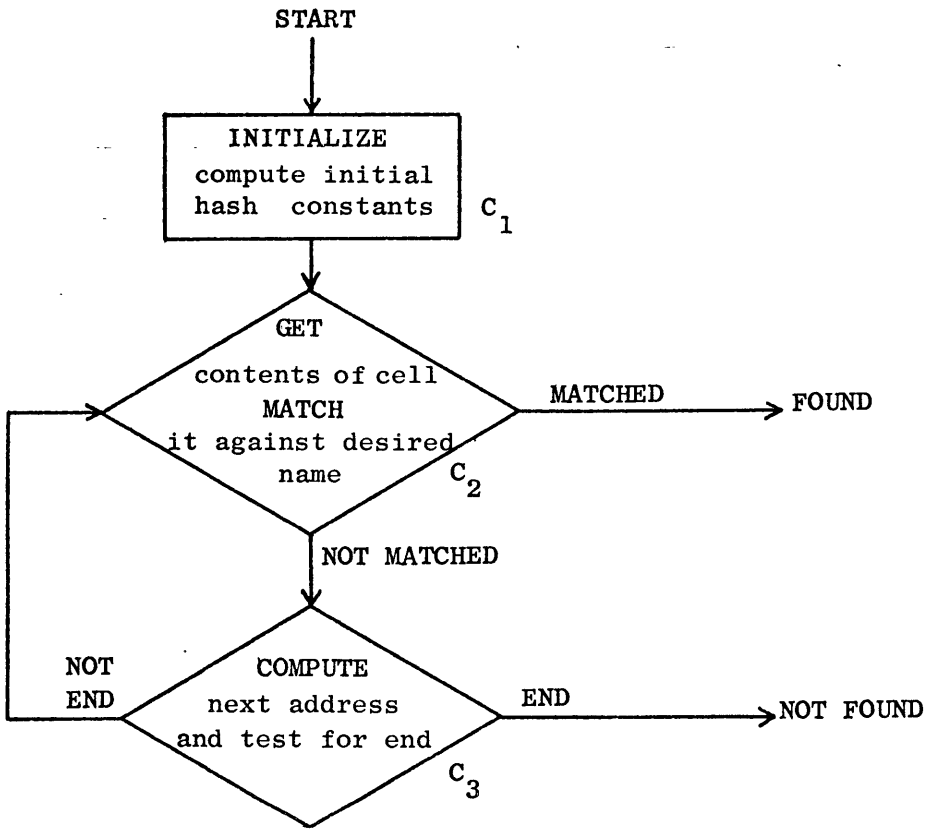


Fig. C.6 Flow diagram of generalized search.

The C_1, C_2, and C_3 in Fig. C.6 are the times required to execute the associated block of the flow diagram.

In terms of these constants, the average access times for linear search and the two types of hash coding described are shown in Table 1.

Table 1

Method	Average search time for expressions included in storage	Average search time for expressions not included in storage
1. linear storage	$C_{11}+C_{21}+((n-1)/2)(C_{21}+C_{31})$	$C_{11}+n(C_{21}+C_{31})$
2. hash-coding chained	$C_{12}+C_{22}+((n-1)/2b)(C_{22}+C_{32})$	$C_{12}+(n/b)(C_{22}+C_{32})$
3. hash-coding index sequence	$C_{13}-C_{33}+(C_{23}+C_{33})(b/n)\ \ln(b/(b-n))$	$C_{13}+(n/(b-n))(C_{23}+C_{33})$

n = number of items stored b = number of hash table locations (buckets)

Derivations of the equations given have been rigorously carried out by Van Horn (1965 and 1966) and others.

"n" is the number of items stored in the memory, "b" is the number of hash table locations. Notice that if only one hash table location is provided, chained hash-coding is the same as a linear storage. Hash table locations are frequently called "buckets", in order to distinguish them from other locations in the computer storage. If one assumes all of the constants C_{ij} in Table 1 have a value of 1, and graphs the average search time against a quantity $\rho = n/b$ one gets a graph that looks like that shown in Fig. C.7. However, it should be noticed that on this scale linear search appears to be just as good as the chained hash-coding and better than the index sequence method. This is caused by the fact that the x-axis is the ratio of items stored to buckets. This appears to be unsatisfactory as a means of comparison, but is presented here because it was used by Van Horn (1966). If we wish to compare performance in another manner we can assume 1000 buckets allocated to both hash coding methods and graph times against number of items stored. That graph is presented in Fig. C.8.

On this set of axes it becomes apparent that the chaining method is far superior, in terms of speed, to linear search. It also appears that it is irrevocably superior to the index sequence method in speed. This speed advantage is achieved by the use of a large amount of list storage space, however, and it is not clear that the small speed advantage gained in the regions where one would normally contemplate using an index sequence method compensate for the fairly large amount of extra storage required by a chaining method.

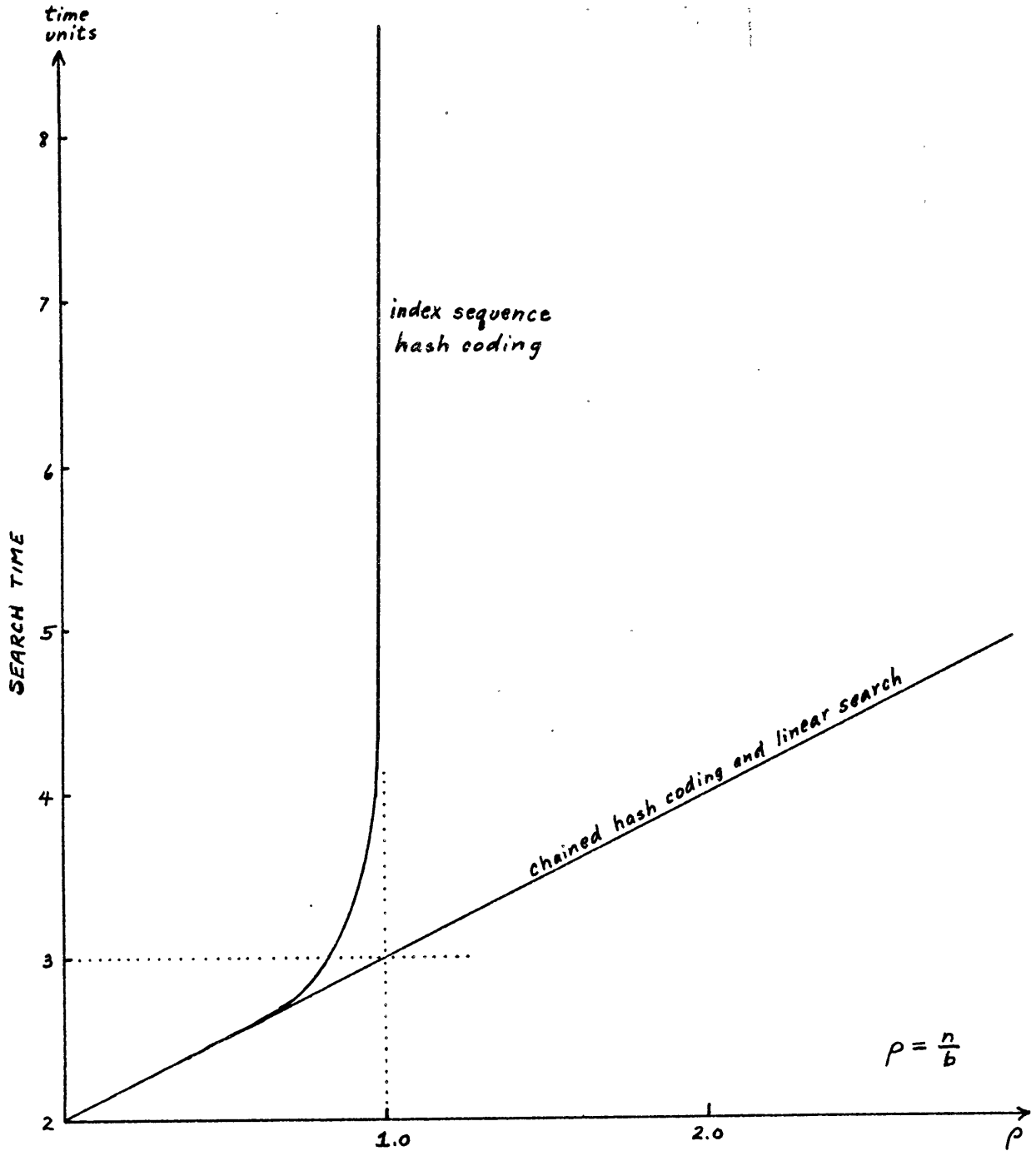


Fig. C.7 Rough graph of average search time against loading ratio ρ .

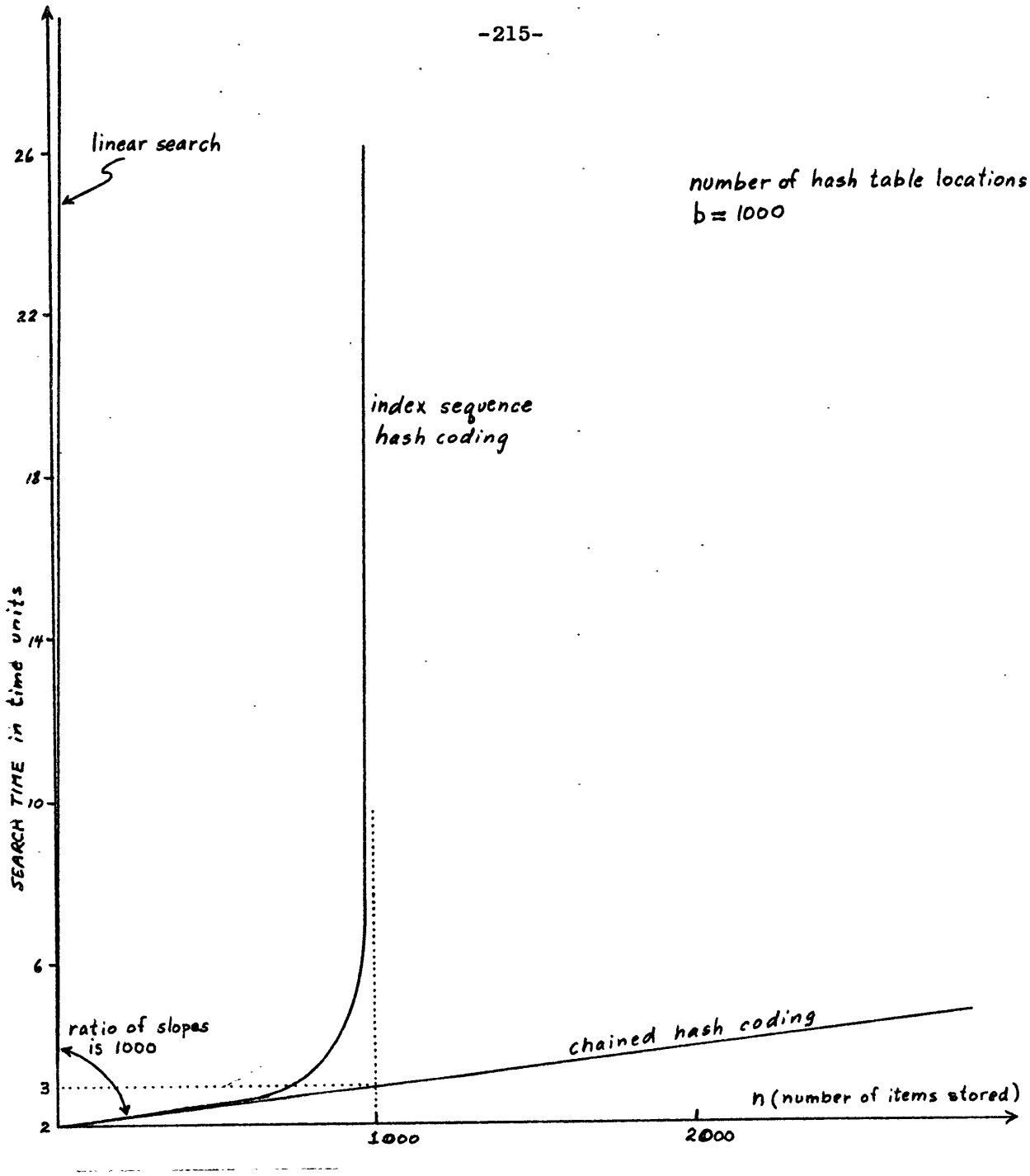


Fig. C.8 Rough graph of average search time against number of items stored, with number of buckets held at 1000.

An idea that immediately arises is: "Why not analyze what happens when total storage for the two hash-coding methods are set equal?" i.e., hash storage for the index sequence method is the same as hash storage plus list storage for the chaining method. To analyze that problem analytically is very unwieldly. Thus, a goal of future research on hash-coding techniques should be to make an experimental comparison of these two mechanisms under such conditions.

Another aspect of the difficulty of comparison is that the constants as given in the earlier tables and later assumed to have value one, have considerably different character. It seems probable that $C_{12} \cong C_{13} \cong C_{33}$ since these constants all represent the computation of a hash integer. It also seems likely that $C_{23} \cong C_{22}$ since the accessing and matching of character strings should be about of the same difficulty in either of the systems. However, C_{32} should be a somewhat smaller time than C_{33} since the former is simply the time for finding the address of the next cell of the linked list instead of the time to generate a new integer. This will have a definite effect on the relationships between the graphs for the two methods. It is difficult, without knowing the exact relation between C_{32} and C_{33}, to analyze how much effect there will be.

Thus, another aspect of any future research on hash coding should be an attempt to measure these various constants for the

two hash coding methods. The knowledge of these constants would give one the ability to predict the performance of each method with amounts of space and data which have not been tested. Such knowledge would also indicate what changes would produce the greatest gains in speed, either by careful programming or by the use of specialized hardware.

APPENDIX D

THE PRODUCTION OF CONTOUR MAPS BY COMPUTER

D.1 Introduction to Contour Mapping

The problem of contour mapping is the following. Given a suitable computer output device and a two-dimensional array of real numbers representing the values of a function at the points of a uniformly spaced grid, to produce a representation of the level curves of that function on the output device. For the purpose of the discussion to follow, only rectangular grids will be considered.

Another problem which frequently is lumped into the contouring problem by those outside of meteorology is the analysis problem. This concerns the production of values of a function of two independent variables at the intersection points of a uniform grid from values of the function at arbitrarily distributed points of the domain. This problem is of considerable theoretical interest in meteorology and is the subject of intensive study and discussion by meteorologists. This problem will not be examined here. Programs to carry out this task by various methods exist at M.I.T. and elsewhere.

Contour maps are produced at most computer installations where meteorological work is carried out. In every instance the programs have several of the following liabilities and deficiencies:

(a) Operates on an array of fixed or restricted size.

(b) Produces a map with fixed proportions and boundaries.

(c) Requires that a contour interval be specified (this is sometimes difficult to do with any success when the array is the result of computation).

(d) Total map completion time too long to allow quick turn-around low-grade usage.

(e) Best possible map quality unsuitable for publication purposes.

(f) No ability to operate in a man-machine interaction environment.

(g) No ability to overlay geographic outlines for spatial reference.

(h) Written in machine or installation dependent forms making general use difficult.

(i) Does not display values of the array as well as the contours.

It seems unlikely that any one contour program operating under the constraints of present day output devices could overcome all of the above listed deficiencies. However, it must be emphasized that unless the library of contour programs at an installation contains at least one program that can minimize or eliminate appropriate subsets of the deficiencies listed for each important type of problem environment, then the contouring problem has not really been solved.

The characteristics by which a contour program is judged must include the following:

(1) Computer time or cost to produce a given map.

(2) User time required to produce the map (turn-around time).

(3) Faithfulness of function representation. This must be judged under some set of assumptions about the function. For example, meteorological functions which have continuous first derivatives would never actually produce a contour line which makes a sharp change in direction.

(4) Precision and completeness of information about the function. For example, spatial reference by geographic overlays or printing of function values at grid points.

(5) Readability. Too little or too much information can make a contour map difficult to interpret.

(6) Ease of use.

Within particular problem environments these criteria will receive varying emphasis. Despite the number of existing contour programs, satisfactory ones exist for only a few environments. Man-machine interaction is one environment which has had no satisfactory contour programs. Another has been batched FORTRAN IV with fast turn-around.

Production of contour maps by computer requires widely varying techniques depending upon the device being used for output. Some of

the methods and devices by which contours can be produced are:

I. CHARACTER PLOTTING - line printer, typewriter or teletypewriter.

II. LINE or VECTOR PLOTTING

(a) ACTIVE PLOTTING - direct view display (i.e. a cathode ray tube CRT or TV like display) which must be refreshed around 30 times per second.

(b) PASSIVE PLOTTING - devices on which map must be plotted once leaving a permanent or semi-permanent outline.

(i) mechanical pen and ink plotters - Calcomp etc.

(ii) cathode ray tube or other electronic means to produce hard copy or film (Stromberg-Carlson 4020). (Usually not directly viewable at the time of plotting).

(iii) direct view CRT storage tube - (Tektronix storage tube).

The most commonly available output device is the line printer and a great deal of contouring has been done on it. This still is the most advantageous device for use in batch processed runs where the output does not require the beauty and precision possible with vector plotting devices. The line printer has the advantages of being available on almost all computers, interspersing the maps directly with other output from the run, being able to print numer-

ical values at grid points with the contours, and of being a cheap, quick and easy way to produce large volumes of low precision information which can be scanned for significant features. Line printer contouring is especially advantageous for debugging or runs where the results are uncertain due to physical or computational instability. This is because of the ability to print values at each grid point and because the time required to produce the contour map is usually entirely independent of the form of the function being contoured. Rough or discontinuous functions will make a character plotting contour map look bad but will not affect computation time, whereas vector plotting methods will produce multitudes of useless contours, taking large amounts of costly time, and not produce printouts of the values at grid points to allow decoding of the cluttered mass of contours.

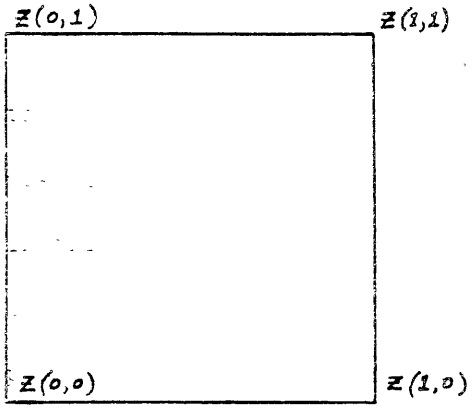
For more beautiful and precise output (to be published or shown to others as final results) at greater cost and longer turn-around time, vector plotting methods become advantageous. These methods allow the overlaying of geographic outlines for physical reference, and are much more satisfying esthetically to the average meteorologist. Thus, vector plotting methods appear most suitable for production of final results, when higher costs and delays can be endured for the sake of a more presentable result.

The only exception to this classification of contouring methods and devices appears to occur in the man-machine interaction (MMI)

environment. There, the turn-around time takes on overriding importance, and the inexpensive typing devices available for output on most consoles are so slow as to be nearly useless for contouring. In this environment the line plotting method, on a direct view cathode ray tube, usually becomes the only suitable approach to producing contours (at least under present hardware capabilities). Costs are high (around $10,000 for an ARDS-II, Stotz 1967 and Gammill 1968b) because of the need for speed. The important problem for contouring in an interactive environment is the production of an easily assimilated map which contains enough information to be useful but not so much as to overload the display (flicker on active CRT's) or require excessive computation time making interaction unfeasible. Thus, production of contour maps in a man-machine interaction environment bears constraints not found in batch processing. For this reason, most presently existing contour programs are unsuitable for use in this environment. The final report of the DISHPAN project (Gammill 1968a), describes some experimental contour programs written for CTSS by the author, and considers some of the problems which are involved.

D.2 Methods of Interpolation

In examining the contouring problem the problem of interpolation must be investigated. Most presently available contour programs make use of a piecewise curvilinear interpolation method, fitting the surface to the values at four points.



Values of function z are assumed to be tabulated at points:

1. x = 0, y = 0
2. x = 0, y = 1
3. x = 1, y = 0
4. x = 1, y = 1

Fig. D.1 Grid rectangle with function valves tabulated at corners.

This gives an equation for the function in the interior of the rectangle defined by the four points of:

$$z(x,y) = z(0,0)\left[(1-x)(1-y)\right] + z(1,0)\left[x(1-y)\right] + z(0,1)\left[(1-x)y\right] + z(1,1)\left[xy\right]$$

$$= z(0,0) + x\left[z(1,0) - z(0,0)\right] + y\left[z(0,1) - z(0,0)\right] + xy\left[z(1,1) + z(0,0) - z(0,1) - z(1,0)\right]$$

This is almost certainly the simplest two-dimensional interpolation method which it is feasible to use, and it does produce a continuous surface which fits the data exactly. However, it is apparent that first derivatives of this surface will normally be discontinuous across the boundaries of the rectangular regions.

Since many (if not most) functions dealt with in the fluid sciences are assumed to have continuous first derivatives, curvilinear interpolation can produce results that look anomalous. This will be seen in the

contour map as a sharp change in the direction of the iso-line as it crosses the boundary between regions.

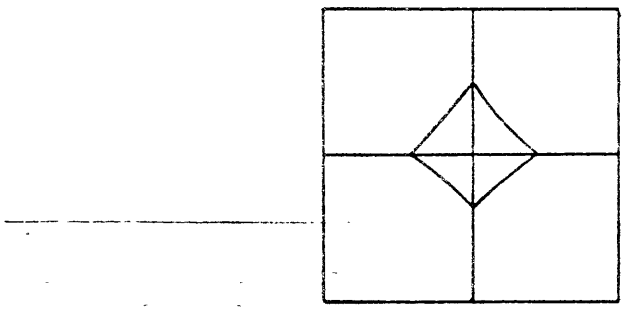


Fig. D.2 Example of the appearance of a contour when the first derivative is discontinuous across boundaries.

Such sudden changes of direction will not be noticeable on maps plotted with a low precision, but become extremely noticeable under higher precision plotting methods.

Therefore, with higher precision plotting it becomes advantageous to use a method of interpolation called two-dimensional piecewise spline interpolation. The salient feature of this method is that besides matching up the values of the function at the boundaries of adjacent regions, it also matches up the first derivatives of the function. To accomplish this, each rectangular region must be fit with third degree polynomials in each direction. To compute the fitting function requires the use of sixteen tabulated values of the function.

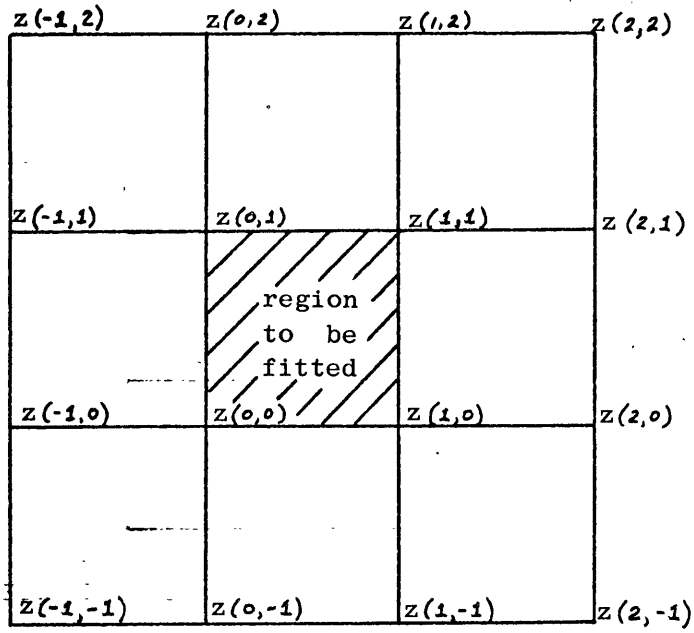


Fig. D.3 Set of grid points used in piecewise spline interpolation.

To create a simplification in the form of the fitting function we define the following, where subscripts indicate the finite approximation to the derivative:

$$z_x(0,0) = \frac{z(1,0) - z(-1,0)}{2} \qquad z_y(0,0) = \frac{z(0,1) - z(0,-1)}{2} \qquad z_{xy}(0,0) = \frac{z_y(1,0) - z_y(-1,0)}{2}$$

$$z_x(1,0) = \frac{z(2,0) - z(0,0)}{2} \qquad z_y(1,0) = \frac{z(1,1) - z(1,-1)}{2} \qquad z_{xy}(1,0) = \frac{z_y(2,0) - z_y(0,0)}{2}$$

$$z_x(0,1) = \frac{z(1,1) - z(-1,1)}{2} \qquad z_y(0,1) = \frac{z(0,2) - z(0,0)}{2} \qquad z_{xy}(0,1) = \frac{z_y(1,1) - z_y(-1,1)}{2}$$

$$z_x(1,1) = \frac{z(2,1) - z(0,1)}{2} \qquad z_y(1,1) = \frac{z(1,2) - z(1,0)}{2} \qquad z_{xy}(1,1) = \frac{z_y(2,1) - z_y(0,1)}{2}$$

also let:

$$a(x) = x^2(x-1) \qquad a(1-x) = -x(1-x)^2$$

$$b(x) = x^2(3-2x) \qquad b(1-x) = (2x+1)(1-x)^2$$

then:

$$
\begin{aligned}
z(x,y) = {} & b(1-x)\left[z(0,0)\,b(1-y) + z(0,1)\,b(y) - z_y(0,0)\,a(1-y) + z_y(0,1)\,a(y)\right] \\
& + b(x)\left[z(1,0)\,b(1-y) + z(1,1)\,b(y) - z_y(1,0)\,a(1-y) + z_y(1,1)\,a(y)\right] \\
& - a(1-x)\left[z_x(0,0)\,b(1-y) + z_x(0,1)\,b(y) - z_{xy}(0,0)\,a(1-y) + z_{xy}(0,1)\,a(y)\right] \\
& + a(x)\left[z_x(1,0)\,b(1-y) + z_x(1,1)\,b(y) - z_{xy}(1,0)\,a(1-y) + z_{xy}(1,1)\,a(y)\right]
\end{aligned}
$$

To provide some intuition as to the manner in which the piecewise spline method of interpolation is able to produce an exact match to function values and finite difference approximations to the first derivative at points of tabulation, the following example is presented. The example shows, in one dimension, how four values positioned at x = -1, 0, +1, +2 along the x axis are fit by piecewise addition of the influence functions produced by the interpolation method. The tabulated values to be fit are y = 1 at positions x = -1 and 0, and y = 0 at positions x = 1 and 2. This forms a unit step function, and tests the ability of the interpolation method at fitting an apparent discontinuity without extreme behavior.

Figure D.4 shows the piecewise influence functions that are multiplied by the value and derivative to be fit and then summed to produce a smooth interpolation function. The top function represents a tabulated value of one at the center point, and the lower function represents a tabulated (finite difference approximated) derivative of one at the center point. Notice that the influence functions

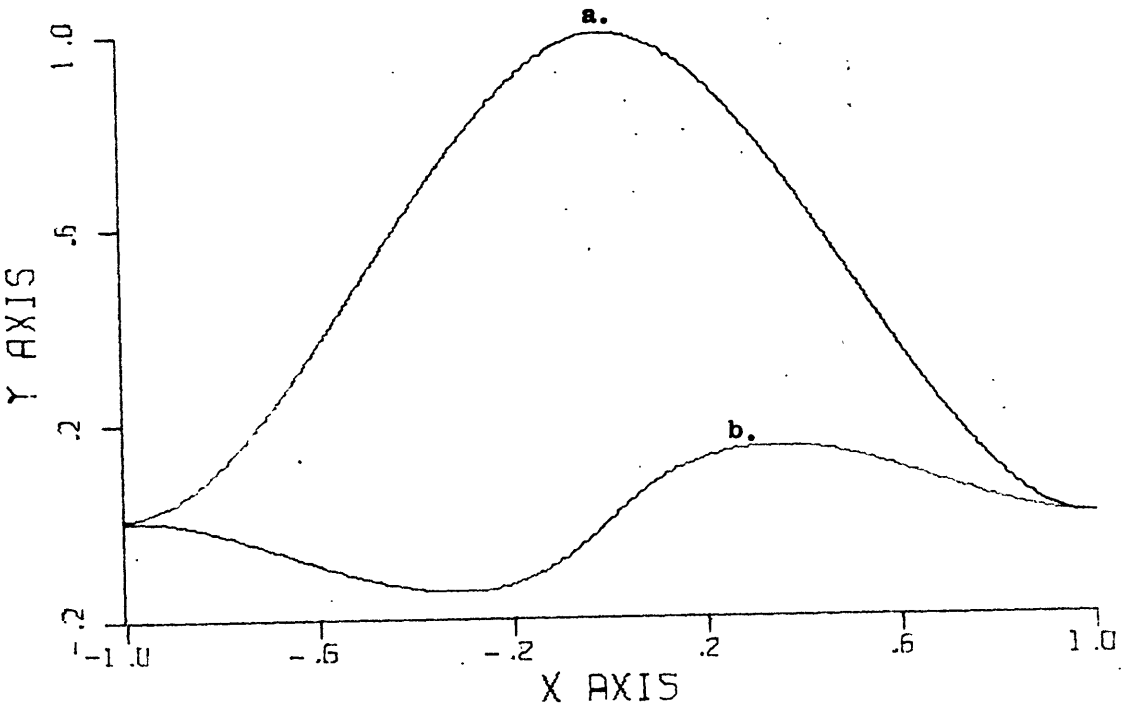


Fig. D.4 Value and derivative piecewise fitting functions

a. value influence function

b. derivative influence function

have a value and derivative of zero at the adjacent points of tabulation (-1 and 1).

Figure D.5 represents the addition of two value influence functions, to partially represent the step function in a smooth, but unesthetic manner. This particular intermediate form may appear to have some merit until one realizes that linearly decreasing tabulated values will produce an undulating interpolation function with a slope of zero at each tabulation point. That case seems pathological enough to keep us from considering this partial form of the interpolation function.

Figure D.6 represents the summing of two piecewise derivative influence functions to produce a partial interpolation function which is due solely to the effect of the derivatives. The centered finite difference approximation to the first derivative of our unit step is $-\frac{1}{2}$ tabulated at $x = 0$ and $x = 1$, and zero elsewhere.

Figure D.7 shows the partial interpolation functions which have resulted from the fit to the values and the fit to the derivatives, as shown in Figures D.5 and D.6 respectively. Figure D.8 shows the result of summing these functions, giving the piecewise spline interpolation function whose values and first derivatives exactly match those given at all points of tabulation. Among the good characteristics of the function plotted in Figure D.8 is continuity of first and second derivatives, good damping on the overshoot caused by the discontinuity of the unit step, and the fact that the function out-

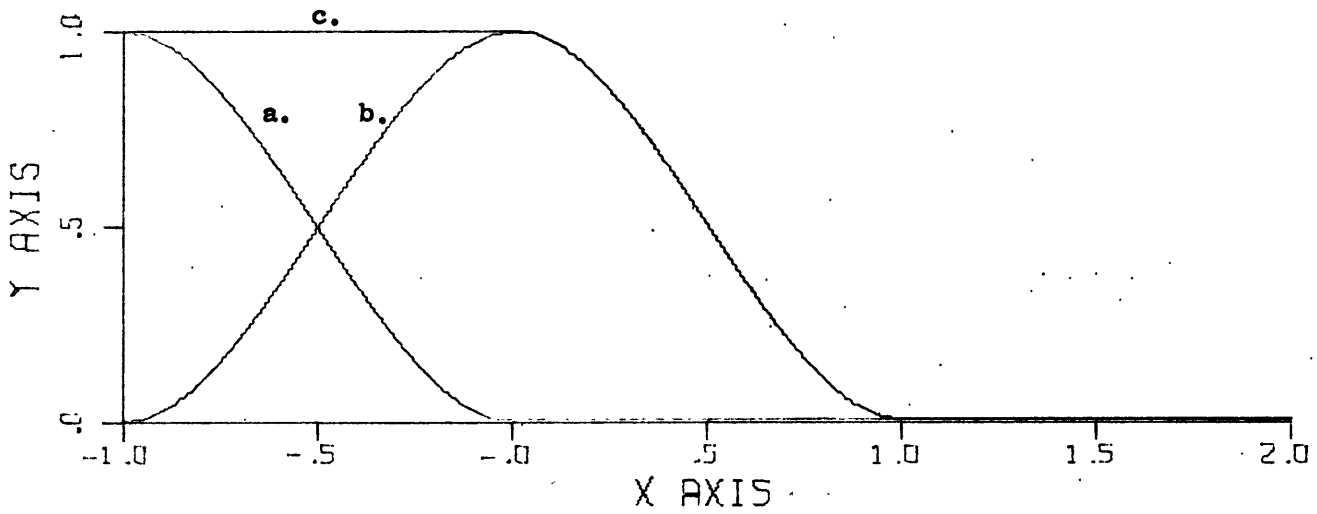


Fig. D.5 Two piecewise value fitting influence functions and their sum

a. influence function for value y=1 tabulated at x=-1.

b. influence function for value y=1 tabulated at x=0.

c. the sum of a and b.

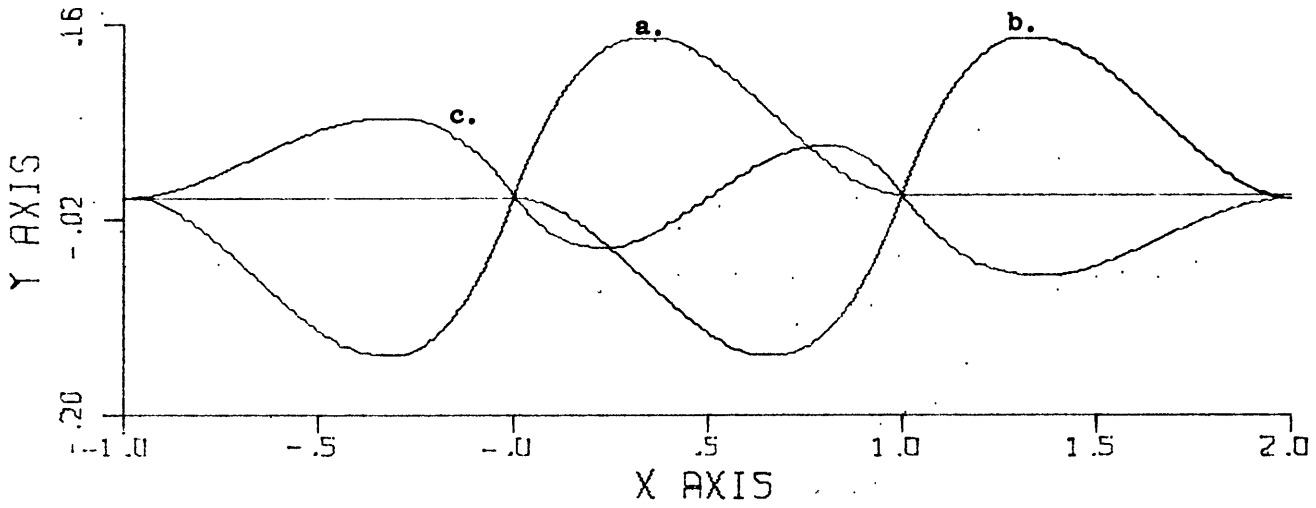


Fig. D.6 Two piecewise derivative fitting influence functions and the result of multiplying each by $-\frac{1}{2}$ and summing.

a. influence function for derivative $\frac{dy}{dx} = 1$ tabulated at x = 0.

b. influence function for derivative $\frac{dy}{dx} = 1$ tabulated at x = 1.

c. the sum of a. and b. after multiplication by $-\frac{1}{2}$.

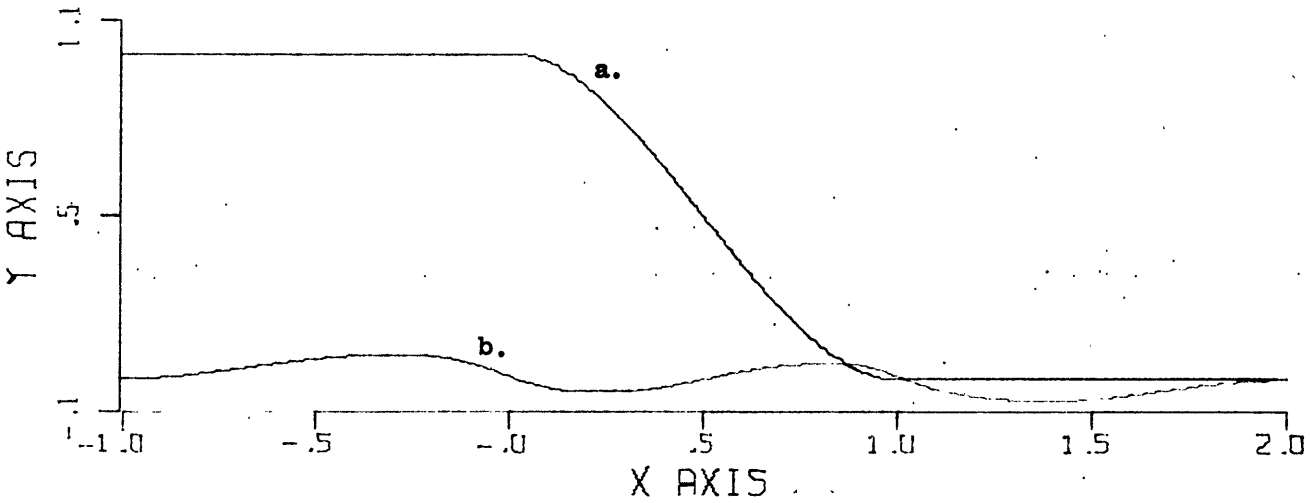


Fig. D.7 Interpolation functions resulting from fitting values (a.) and derivatives (b.).

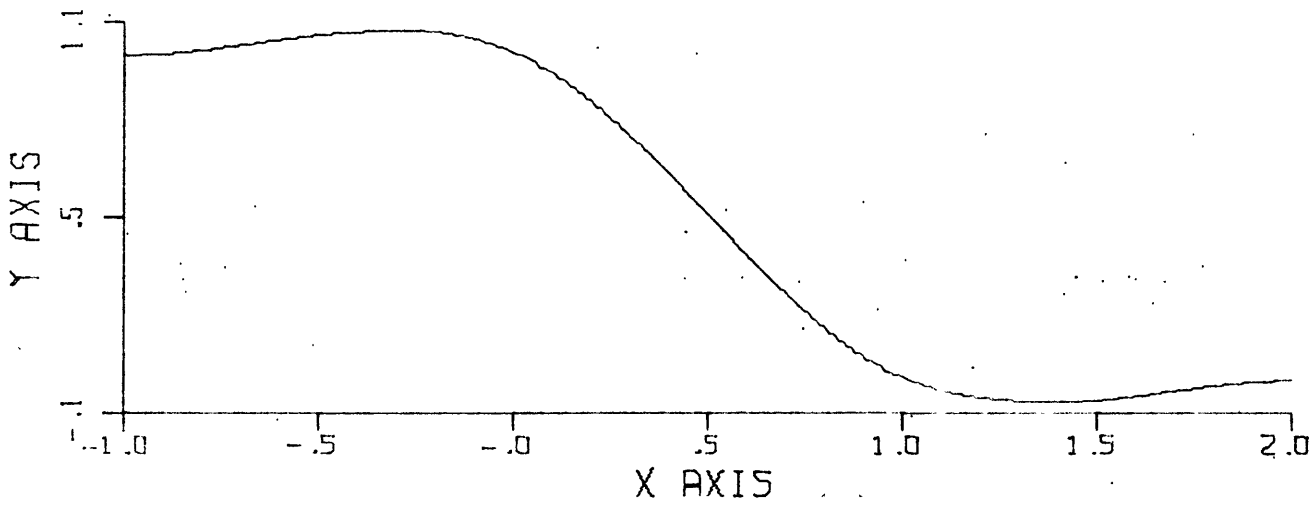


Fig. D.8 Completed piecewise spline interpolation function resulting from fit to step function.

side of the region shown will be completely unaffected by this discontinuity.

A particular advantage of the spline interpolation method is its ability to interpolate in regions adjacent to a boundary, where not all the points are available. This is accomplished by computing the uncentered difference forms of the derivatives. It is interesting to note that if only one region comprises the whole map (i.e. only 4 values tabulated at the four corners of the map are given) the spline interpolation method reduces to the curvilinear method.

The derivatives become:

$$Z_x(0,0) = Z_x(1,0) = Z(1,0) - Z(0,0) \qquad Z_y(0,0) = Z_y(0,1) = Z(0,1) - Z(0,0)$$

$$Z_x(0,1) = Z_x(1,1) = Z(1,1) - Z(0,1) \qquad Z_y(1,1) = Z_y(1,0) = Z(1,1) - Z(1,0)$$

$$Z_{xy}(0,0) = Z_{xy}(0,1) = Z_{xy}(1,0) = Z_{xy}(1,1) = Z(0,0) + Z(1,1) - Z(0,1) - Z(1,0)$$

Using these versions of the derivatives in the spline interpolation equation produces:

$$\begin{aligned} Z(x,y) = {} & Z(0,0)\left[\left(-a(y)+a(1-y)+b(1-y)\right)\left(-a(x)+a(1-x)+b(1-x)\right)\right] \\ & + Z(0,1)\left[\left(a(y)-a(1-y)+b(y)\right)\left(-a(x)+a(1-x)+b(1-x)\right)\right] \\ & + Z(1,0)\left[\left(-a(y)+a(1-y)+b(1-y)\right)\left(a(x)-a(1-x)+b(x)\right)\right] \\ & + Z(1,1)\left[\left(a(y)-a(1-y)+b(y)\right)\left(a(x)-a(1-x)+b(x)\right)\right] \end{aligned}$$

and noticing that

$$-a(x) + a(1-x) + b(1-x) = 1 - x$$

$$a(x) - a(1-x) + b(x) = x$$

We find that we do indeed have the curvilinear interpolation equation.

By now it should have become clear that the use of the finite difference approximation to the derivative is entirely optional. If one has access to actual values for the derivatives, then those may be used. Furthermore, if some special constraint on the resulting function is desired, that constraint may be reflected in the method by which the tabulated values of the derivatives are computed. For example, in contouring a grid whose values represent concentrations of pollutants, one might have large regions of zero values. When such a region is adjacent to a region of positive values, a step function problem much like that shown in Figures D.3 to D.8 will occur. The result, if the centered finite difference approximation to the derivative is used, will be an undershoot of the fitted function into negative values (as shown in Figure D.8). A negative pollutant concentration seems somewhat anomalous, so one might wish to prevent this result by requiring tabulated values of derivatives to be zero at all points where the tabulated function values were zero.

The piecewise spline interpolation method was independently derived by the author, although the general method is not new. References to generalized spline interpolation methods have been subsequently

discovered (Greville 1967), but a literature search has not been undertaken to discover if this particular method has been described.

D.3 Methods of Contour Plotting

The two subsections, which follow, will present the two major methods by which contour maps may be plotted. These methods may be characterized as character plotting and vector plotting.

D.3.1 Character plotting methods of contouring

Two basic methods of producing contour maps using character plotting devices have developed over the years. These can be characterized as filling (or shading) the regions between contours and filling the character positions through which contours pass. Examples of each are shown in figures at the end of this appendix.

The method of filling regions between contours operates by dividing the value of the function (computed by an appropriate method of interpolation) at the center of a particular print position by the contour interval to produce an index number indicating what character should be printed there. Characters to be used for filling are usually carried in a table, and the index number must be converted to a subscript, referencing the table, by taking it modulo the size of the table and adding one. This method is undoubtedly the simplest available, and has the singular advantage that the algorithm's execution time is completely independent of the form of the function contoured. A contour program using this method has been written by the author

(Gammill, 1968c) and is in frequent use at the M.I.T. Computation Center.

The method of filling character positions through which contours pass is only slightly more complicated than the preceding. Each print position is characterized by the diagram shown in Figure D.9

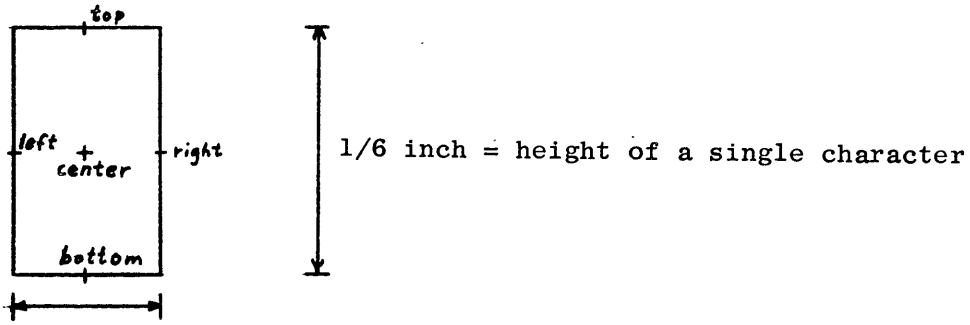


Fig. D.9 Diagram of a print position.

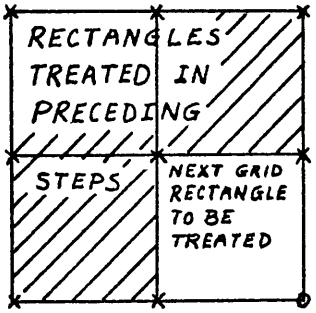
The value of the function is computed (via interpolation) at the positions marked top, bottom, right and left. If the values at top and bottom, or right and left straddle a contour value, then the character which signifies that contour is printed in this position. Otherwise a blank is always printed. This method can be programmed to gain a speed advantage from the fact that most print positions will normally be filled with blanks. This makes the execution time of the program dependent upon the form of the function. However, this is of little concern in this case since the upper bound of the time required can always be kept to that time required to fill every print position with a non-blank character.

The method of filling regions between contours is the more popular of the two methods. This appears to stem from the fact that as contours become closely packed together this method allows more blank space for easier distinguishability of the contour position. In fact, if the contours are evenly spaced and the method of filling is such as to produce consecutive blank and non-blank regions, then there will be approximately equal amounts of blank and nonblank space. This is unlike the method of filling contour passage positions where, as contours become close together, the region becomes a solid mass of characters. A disadvantage of the method of filling regions between contours is that a filled region may seem to disappear if it becomes narrow enough to fit between print positions (i.e. if the contours become very closely packed). In a similar situation the other method produces an overlapping of the characters signifying the passage of a contour, but this overlapping can usually be easily untangled by examination, so long as the packing does not become too extreme.

Once a character plotting method has been chosen, the problems of choosing a method of interpolation and a form of presentation arise. We have presented the two most suitable interpolation methods, but we have not mentioned the fact that for line-printer maps it is often advantageous to produce a map as several separate strips to be taped together upon completion. This technique allows the production of much larger maps, when desired, but it is obviously not usable on

any device which does not produce hard copy, or where the copies are not large enough to be joined accurately.

Another kind of question in character plotting methods is whether speed of operation or ease of programming is to be emphasized. If ease of programming is emphasized then it is most advantageous to write a program where each grid rectangle is treated in exactly the same way, operations at the grid points (such as calculation of the finite difference approximations to the derivative for the spline method) being carried out at all four corners for every rectangle, and interpolations within the rectangle proceeding in one step from the values at the grid points. However, if speed of program operation is of prime importance, this can be achieved by recognizing that interior grid rectangles share three grid points and two sides with rectangles treated in preceding steps.



x Marks grid points where values have previously been computed.

o Marks the grid point for which values must now be computed.

Fig. D.10 Grid point calculation sequence diagram.

This means that only one grid point per interior rectangle has not been utilized in some previous computation. Thus, by saving the results of earlier grid point calculations, labor can be decreased

by nearly a factor of four. Another method of achieving speed is to carry out interpolations in two steps. If the program makes a sequence of interpolations for different character positions along a row, it is quicker to carry out a vertical interpolation at each side of the grid rectangle and then carry out individual horizontal interpolations to the character positions along the row within the rectangle. Further, since the vertical interpolation value is shared by two rectangles, a right side vertical interpolation from one rectangle can be used as the left side vertical interpolation for the next. Thus, for an interior rectangle, each row of character positions requires one vertical interpolation and each interior character requires one horizontal interpolation. An example of this is shown in Figure D.11.

D.3.2 Vector Plotting Methods of Contouring

Vector plotting methods for producing contour maps require much more decision making by the program than do character plotting methods. For each grid rectangle the program must decide what contours pass through, where they enter and leave, and how they behave in the interior. Numerous pathological situations can arise, and it is difficult to write a program that can handle them all successfully. In general it cannot be guaranteed that a vector plotting contour program will run to completion in a given amount of time, for these programs are highly input dependent. That is, the form of the function contoured has a direct effect on the amount of time

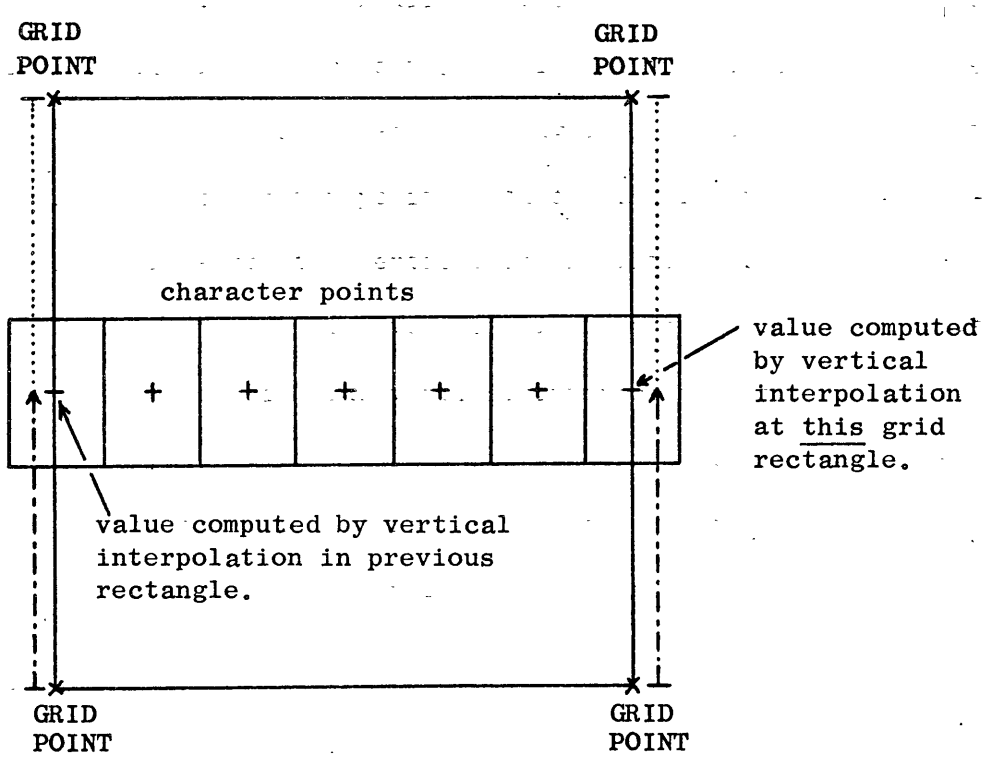


Fig. D.11 Diagram of grid square with row of print positions.

For this row of characters the labor for interpolation can be reduced from 6 complete two-dimensional interpolations to 6 one-dimensional interpolations (1 vertical and 5 horizontal).

required to produce its contour map. Two functions having the same range, but with one having wild excursions to the extremes of that range, will require much different amounts of labor. Thus, vector plotting programs might well be characterized as being inherently less stable than character plotting contour programs, because of input dependence.

One of the troubles which immediately confronts the writer of a vector contouring program is the problem of what to do when a grid point value (or collection of grid point values) is exactly equal to the value of a contour. One solution to this problem, which is simple although perhaps not elegant, is to treat all such equalities as inequalities in some fixed sense.

The standard method for carrying out vector contouring is to treat each grid rectangle as having sides which do not have end points. This constrains each side to be shared by no more than two rectangles. Since contours will only be allowed to enter or leave a rectangle by a side (and not through a grid point) we are assured of being able to tell which adjacent rectangle a contour will be found in next.

Other standard methods for carrying out vector contouring are the use of linear interpolation along sides to find the position of the contour crossing, and the use of straight lines to connect side crossings to make a contour. Although these methods may not be the most elegant or esthetically pleasing possible, they do allow vector

contouring to be carried out in a relatively simple manner. The consequences and difficulties which arise when these methods are not used will be discussed further on.

As a direct consequence of the use of linear interpolation, any contour value has at most one crossing point on any grid rectangle side. Further, the four sides of any rectangle will be crossed exactly zero, two or four times, by any contour value. This gives us exactly nine different cases which can occur for any grid rectangle with respect to a particular contour value (assuming that we ignore the directionality of the contour line). The nine cases are shown in Figure D.12.

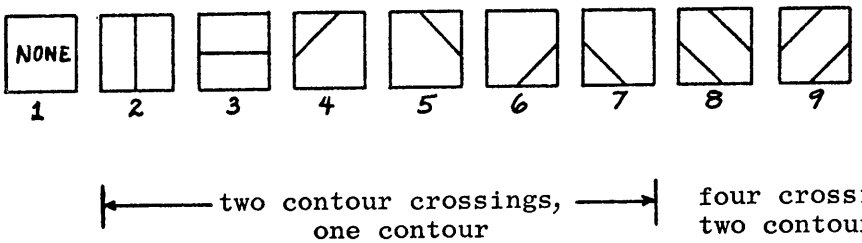


Fig. D.12 Nine possible cases of a contour crossing a grid rectangle using linear interpolation.

If we do not ignore directionality, i.e. if we recognize the direction of the function gradient with respect to the contour, then each case where contours are present doubles and there are seventeen cases. This fact becomes of interest when the method of proceeding through the grid must be decided upon. The simplest program can be written by attacking each grid rectangle in a fixed order, and carrying the calculations and plotting required to completion,

before moving on to the next rectangle. Although simple, this method has some serious deficiencies for certain kinds of environments. In particular, if a pen and ink plotter is being used to produce the final result, the time required to put the pen down and pick it up in drawing each short line segment becomes significantly expensive. Even for an electronic display, two display commands must usually be given for each line segment, where one would suffice if the contour were followed in a continuous manner. Also in the pure "rectangle at a time" approach, the program must carry out redundant interpolations.

If the programmer begins to be concerned about redundant interpolations in the pure "rectangle at a time" method, he quickly comes upon the idea of saving the results of interpolations until the rectangle which shares the side is processed. The next step is to notice that so long as the contour moves in the correct direction, the redundant display commands can be eliminated also. At this point, the program can no longer be considered to use a true "rectangle at a time" method, but has become a hybrid of that method and the one we will discuss next.

A more efficient method, although considerably more complicated, is that of following the contour. This method keeps the number of display commands to a minimum and also tends to eliminate redundant interpolations. The method of contour following used by the writer differs significantly from that used in any other known contour

program, and will be discussed in slightly greater detail.

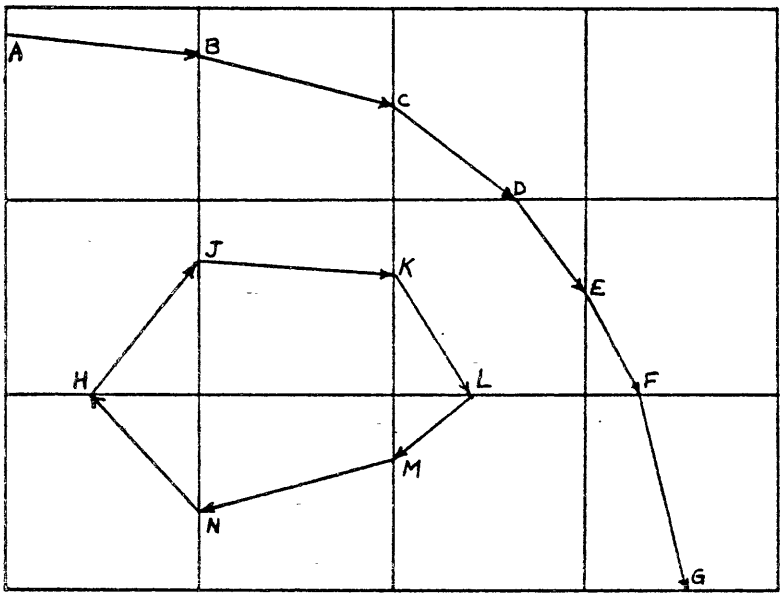
An important fact, noted during the development of programs that follow the contour, is that a contour line, viewed as a collection of directed line segments connecting rectangle edge crossing points, is strongly analogous to a linked list. Each crossing point has a predecessor and a successor, unless it is at the edge of the grid. In that case, if it is a point of contour entry it has only a successor and if it is a point of contour exit it has only a predecessor. This analogy leads logically to the development of a program which produces contours in two passes. The first pass is done by a rectangle at a time method, interpolating to find the positions of individual directed line segments. However, instead of plotting those segments, they are placed in the correct orientation into a linked list structure. At the completion of the pass, the list structure provides a complete description of the contour map, but in a form that is ideally suited for following each contour.

This linked list method compares very favorably with methods normally used in contour following (Washington 1968). Those methods usually begin by searching the grid for a contour crossing, and then follow that contour until it terminates by reaching a side, or closing on itself. During the following process a flagging procedure of some sort must be carried out, so that if this contour is discovered again at some later point in the search process, it will be recognized as being completed. This approach has several disadvantages. The alter-

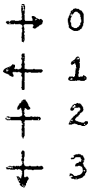
nation between searching and following tends to make the search procedure inefficient. The storage required for flagging is directly proportional to the size of the grid array being contoured (and that can be very large). Finally, at no time in the contouring process is a concise and complete data structure representing the layout of the contours produced. The linked list method carries out all searching and interpolation (but no following) in the first pass. The resulting linked list could be unraveled to form a very compact model of the contour map, and the size of the linked list is proportional to the number of line segments to be plotted rather than the size of the grid.

The usefulness of the list structure as a model of the contour map has not, as yet, been exploited for anything except ordering the crossing points for efficient plotting. In the future this list structure model may prove useful in allowing the grid to be dynamically mapped onto unevenly spaced or moving grids or allowing certain contours to be redrawn or modified interactively. Such "display models" have proved to be of high utility in on-line graphical interaction systems aimed at other areas, and that would seem likely to be true here as well. Figure D.13 shows an example of a simple contour map, and its linked list structure model.

The looks of a contour map produced by linear interpolation along grid rectangle sides, points being connected by straight lines, can be improved somewhat by adding some further structure



EDGE CROSSING DIRECTION CODES



	direction code	value	pointer
H	2	x	J
J	0	y	K
K	0	y	L
L	3	x	M
M	1	y	N
N	1	y	H

list structure for closed contour.

	direction code	value	pointer
A	0	y	B
B	0	y	C
C	0	y	D
D	3	x	E
E	0	y	F
F	3	x	G
G	3	x	END

list structure for contour with entry and exit.

Fig. D.13 Contour map and linked list structure model.

within the grid rectangle. Figure D.14 indicates how this may be done (from IBM E20-0117-0).

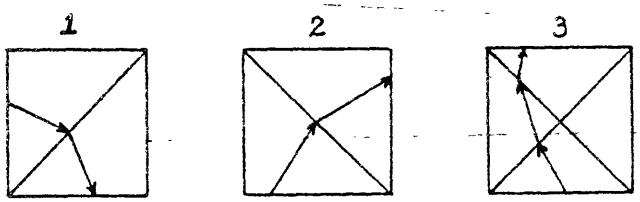


Fig. D.14 Diagonal linear interpolations to produce smoother contours.

Diagonal interpolations may still be carried out by linear interpolation, allowing single contour line segments to be broken into two or more segments. In the case where two diagonals are used, a center point value may be computed as the mean of the surrounding points, and used in interpolations to each of the corner points. Following the contour through the interior of the grid rectangle becomes somewhat more complicated using these methods, but the smoothness of the contours produced is improved.

When interpolation methods are used which are of higher degree than linear, the problem of following a contour within the interior of a grid rectangle becomes much more difficult. Instead of a maximum of one crossing by any contour value on a rectangle side, the number of crossings can be as high as the degree of the interpolation formula. This means that our piecewise spline interpolation method can produce three crossings per side, or a maximum of twelve exit and entry points per grid rectangle. An example of a grid rectangle with

twelve exit and entry points by a particular contour value is shown in Figure D.16. In the same figure it can be seen just how complex the configuration of contours can become within a given rectangle. This complexity is the primary reason why most vector contouring schemes stay with linear interpolation along edges and straight lines between edge crossings.

One example of the complexity introduced by nonlinear interpolation is the problem which arises when interpolating along the side of a grid rectangle using our 3rd degree piecewise spline interpolation method. Since the equation is 3rd degree, the finding of all roots can be quite complicated, and we are only interested in real roots between zero and one. Two relatively simple iterative methods of finding roots suggest themselves. The first involves dividing the interval up into small segments, computing function values at segment ends, and finding a segment whose end values straddle a root. Once such a segment is found, the position of the root in that segment can be approximated by linear interpolation. Finding such a segment can be sped up by first finding a likely segment by linear interpolation, then if unsuccessful, moving towards more likely segments by using the slope and value of the function over the segment to choose a direction of search. The second iterative technique is the Newton-Raphson method. This method uses iterative approximation to find a root. The value at the n-th step is (C being the contour value):

$$x_n = x_{n-1} + \frac{C - f(x_{n-1})}{\frac{\partial f(x_{n-1})}{\partial x}}$$

No matter what method is used, it must be remembered that if linear methods indicate one contour crossing, then by 3rd degree interpolation there can be one or three roots, and if linear methods indicate no crossings there may actually be two roots. It should also be noted that of the two influence functions that make up the piecewise spline interpolation method described in Section D.2, the value influence function will produce exactly as many roots as linear interpolation, while extra root pairs (if any) will always be produced only by the effect of the influence function for representing slopes.

If one is adamant about the need for smoother, more esthetic looking, contour maps, the complexity of the interior of a grid rectangle must be dealt with in some effective manner. The conceptually simplest approach to solving the problem is to define a finer (micro-scale) grid mesh on the interior of the original (macro-scale) grid rectangles. To obtain the values of the function on this finer grid mesh we use a nonlinear interpolation scheme (such as the piecewise spline described earlier). Then, to follow the contour through this micro-scale grid, we utilize linear interpolation and straight line segments. The smoothness of the contour map (i.e. the shortness of the line segments) can be controlled by setting the fineness

of the interior grid. Increases of fineness will, of course, increase execution time dramatically since the program must draw a complete micro-scale contour map within each macro-scale grid rectangle through which a contour passes. Furthermore, the fact that non-linear interpolation can produce closed contour figures completely interior to a macro-scale grid rectangle means that it is not easy to tell whether a contour passes through a particular section of micro-scale grid, except by searching it completely. This means that attempts to optimize micro-scale grid contouring, by only carrying out non-linear interior interpolations between macro-scale entry and exit points, can cause us to completely miss the presence of an interior contour which has no entry or exit. For example, in Figure D.15, if all interpolations are carried out at micro-scale grid points, 96 non-linear interpolations are required. If instead, only interpolations required between the macro-scale entry and exit points are carried out (at micro-scale grid points of Figure D.15 marked by small o's), then 30 non-linear interpolations are required. However, doing that means that the small closed contour will be completely missed. This means that if one wants to produce all submacro-scale closed contours (those interior to a grid rectangle), either complete interpolation and searching must be carried out on all micro-scale grids, or else an effective means must be found for calculating upper and lower bounds of the values which will be found on any micro-scale grid, or subsections thereof. Upper and lower bounds can be used to decide

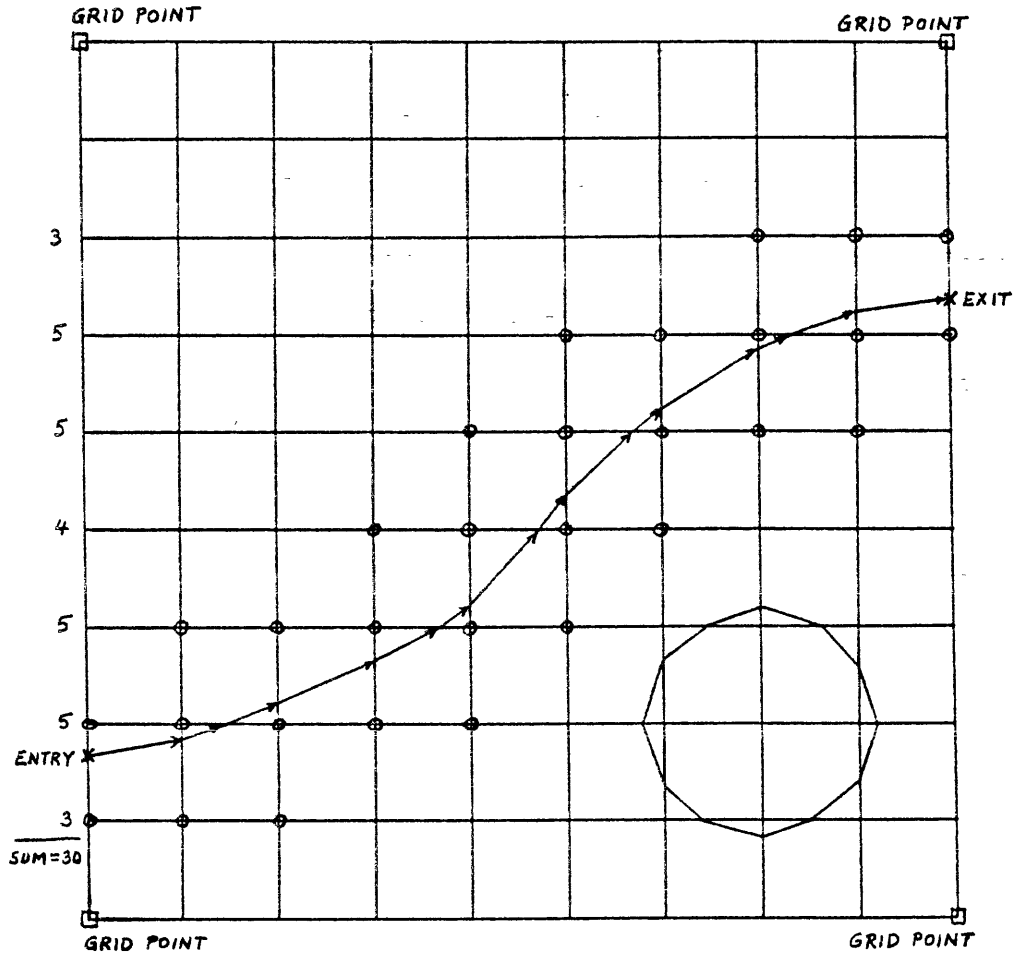


Fig. D.15 Example of a micro-scale grid within a grid rectangle.

if a contour could exist within the micro-scale grid or its subsection. If a contour could exist, interpolation and searching must take place over the entire grid or section.

Just as the method just described is a 2-dimensional extension of the 1-dimensional segmented interpolation method described earlier, the Newton-Raphson method of iterative approximation has a two dimensional counterpart. It will be presented here primarily for purposes of completeness, for though it offers potentially greater efficiency and accuracy than the micro-scale grid method, some of the decision problems which it creates appear well nigh insoluble. The two-dimensional N-R method of following a contour must be initialized by giving it a contour entry point to a grid rectangle. The method then computes the gradient of the function at that point, and moves into the grid rectangle (perpendicularly with respect to the gradient) by some fixed step size. The equations for this step are:

$$\Delta x_s = -\delta \left[\frac{\frac{\partial f}{\partial y}}{|\nabla f|} \right]$$

$$\Delta y_s = \delta \left[\frac{\frac{\partial f}{\partial x}}{|\nabla f|} \right]$$

The point reached will (if δ is chosen small enough) be quite near the contour. A point which is exactly on the contour can then

be found by two-dimensional Newton-Raphson approximation. The iteration equations are:

$$x_{n+1} = x_n + \frac{\frac{\partial f(x_n, y_n)}{\partial x}\left[C - f(x_n, y_n)\right]}{\left(\frac{\partial f(x_n, y_n)}{\partial x}\right)^2 + \left(\frac{\partial f(x_n, y_n)}{\partial y}\right)^2}$$

$$y_{n+1} = y_n + \frac{\frac{\partial f(x_n, y_n)}{\partial y}\left[C - f(x_n, y_n)\right]}{\left(\frac{\partial f(x_n, y_n)}{\partial x}\right)^2 + \left(\frac{\partial f(x_n, y_n)}{\partial y}\right)^2}$$

Here C is the contour value. The method proceeds along the contour, alternately stepping forward and then refinding the exact position of the contour, until it reaches the other side of the grid rectangle.

Some problems which make this method difficult to use are the following:

(1) What to do when the gradient becomes zero?

(2) How to prevent the δ step from taking us across a saddle when two contours of the same value approach very close to one another, but do not touch?

(3) How to find closed contours in the interior of a grid rectangle?

Such problems are prevented in the micro-scale grid method by the inherent errors, which make consistent (although arbitrary) decisions about the contour configuration.

D.4 Miscellaneous Problems in Contouring

A number of associated calculations pertain to contour mapping. One of these, the calculation of a contour interval, can be of great use in making contouring more convenient and easy to carry out.

Most contour programs in existence require the user to specify, in one way or another, a sequence of values for which he desires contours to be produced. If those values are evenly spaced, the distance between them is called the contour interval. In many cases, where the range of values and limits of the gradient are well known to the user, this is not hard to do. However, in some cases, eg. when the grid to be contoured has resulted from a numerical model which has unknown characteristics, the necessity of specifying a contour interval or sequence of contour values can cause useless results to be produced (i.e. the entire range fits between contour values, or so many contours occur as to be unreadable). The following is an extremely useful algorithm for internal specification of a contour interval. Integer powers of ten are always contour values, and the contour interval is such as to produce no more than twice as

many contour levels as the basic number specified:

N = basic number of contour levels specified

Range = Max (values on grid) - Min (values on grid)

Diff = Range/N — This is the rough (high) estimate of the contour interval, which must be rounded down to a suitable value.

ILOG = integer value of LOG_{10} (Diff)

POWER = 10^{ILOG}

CONST = DIFF/POWR rounded down to the nearest of (1.0,2.0,2.5,5.0).

CONTOUR INTERVAL = CONST x POWER.

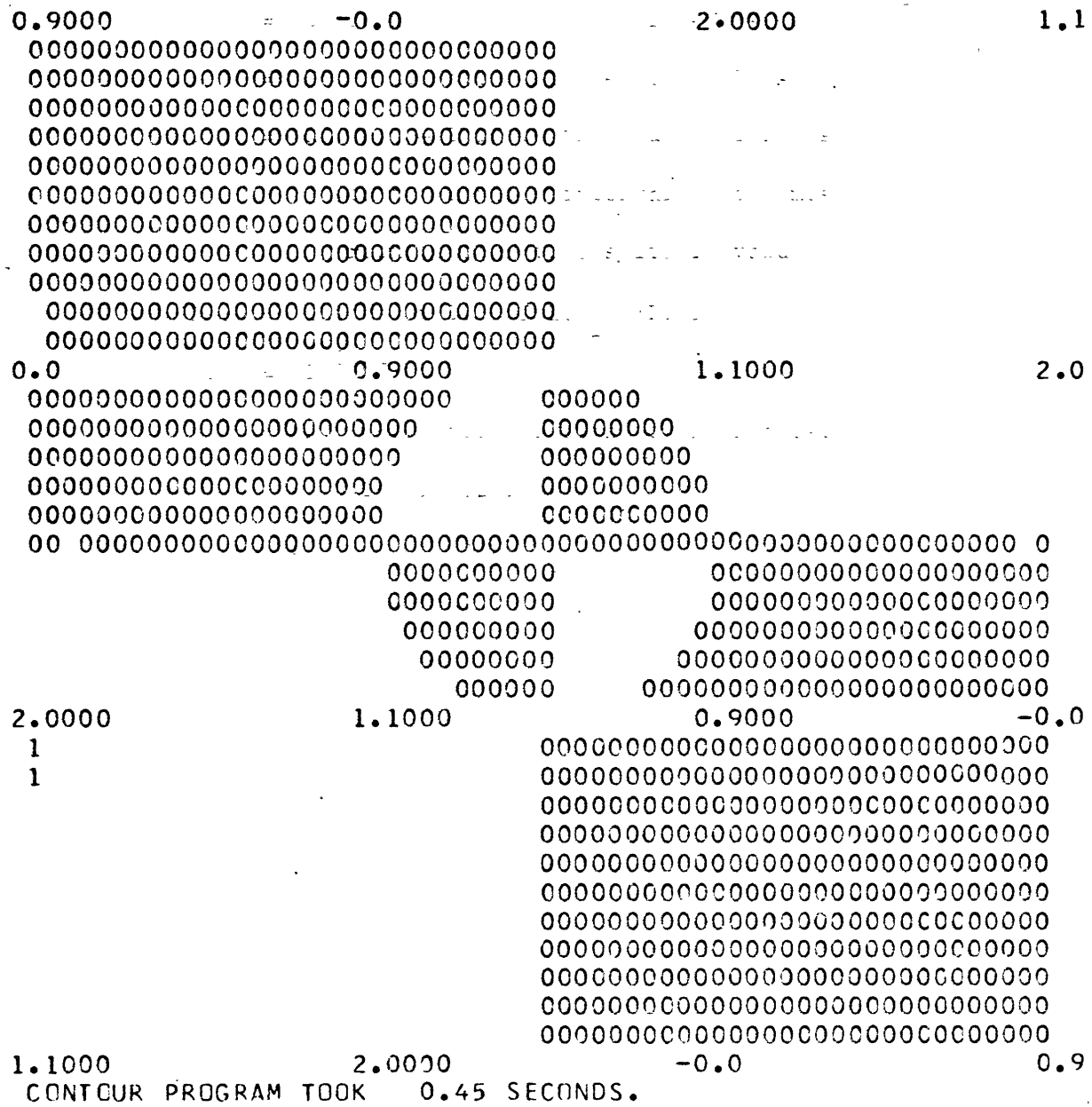


Fig. D.16 Contour map showing complex shapes possible within a grid rectangle when third degree interpolation is used.

Fig. D.17 Contour map, using method of filling regions between contours, showing smoothness produced by third degree spline interpolation.

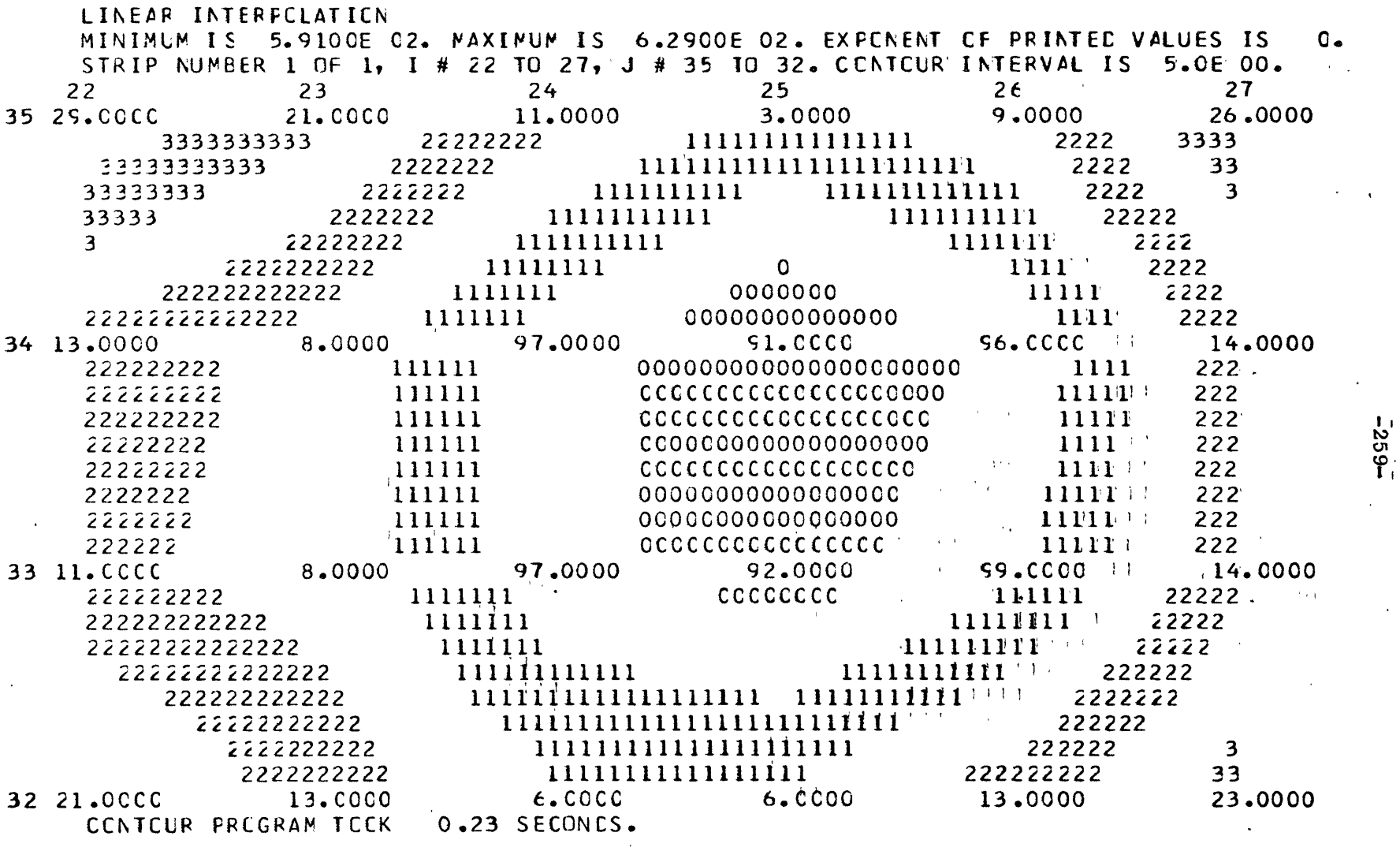
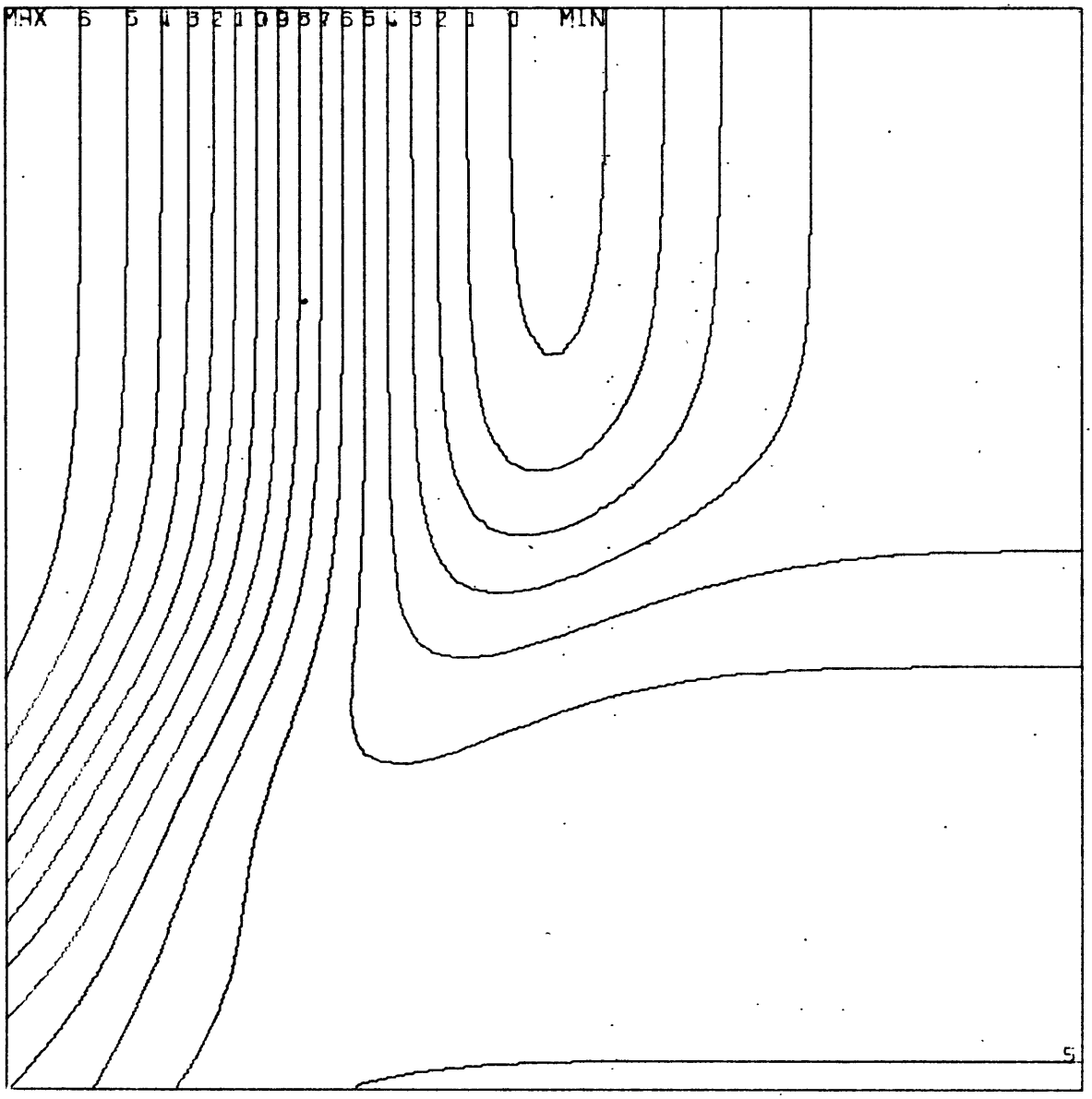


Fig. D.18 Contour map of the same data used in Fig. D.17, now using linear interpolation. Note sharp changes in contour direction at rectangle edges.

```
     LINEAR INTERPOLATION
     MINIMUM IS  5.9100E 02. MAXIMUM IS  8.1100E 02. EXPONENT OF PRINTED VALUES IS   0.
     STRIP NUMBER 1 OF 1, I # 14 TO 30, J # 37 TO 27. CONTOUR INTERVAL IS  2.0E 01.
    14   15   16   17   18   19   20   21   22   23   24   25   26   27   28   29   30
37 11.0 79.0 50.0 37.0 27.0 18.0 15.0  8.0  2.0 90.0 78.0 71.0 71.0 81.0  2.0 20.0 35.0
      555   44444         333333333333         2222222222       2222222    333333
       555    4444         33333333           22222222                22222     33333
36  1.0 69.0 42.0 27.0 17.0 -0.0 88.0 76.0 63.0 51.0 39.0 33.0 34.0 50.0 73.0 95.0 15.0
     555    444       3333333      22222222       1111111111111111111    22222    33
     555   4444     33333333     222222       1111111111           111111    22222
35 89.0 59.0 32.0 15.0  3.0 83.0 61.0 45.0 29.0 21.0 11.0  3.0  9.0 26.0 48.0 70.0 97.0
      5    444      3333333     2222      11111111             0          11111    2222
      5   4444    3333333     2222    11111111             CCCCCCCC      11111     222
34 76.0 49.0 23.0  8.0 92.0 65.0 38.0 23.0 13.0  8.0 97.0 91.0 96.0 14.0 34.0 53.0 80.0
       4444     3333333    2222    111111              000000000000      111111    222
      4444     3333333     222    111111              00CCCCCCCCCC      111111     22
33 54.0 35.0 18.0  5.0 85.0 56.0 33.0 18.0 11.0  8.0 97.0 92.0 99.0 14.0 29.0 45.0 68.0
     44       3333333      222     1111111               CCCCCCCC          1111111     2
              33333333     2222    11111111                                11111111    2
32 32.0 20.0 10.0 -0.0 88.0 68.0 46.0 31.0 21.0 13.0  6.0  6.0 13.0 23.0 29.0 43.0 65.0
       3333333333333         22222      11111111                   111111111111     2
     33333333333333333         22222      11111111               1111111111111    2
31 13.0  8.0  5.0  2.0 -0.0 90.0 71.0 53.0 36.0 21.0 14.0 18.0 26.0 31.0 34.0 43.0 65.0
     333333333333333333333         22222      111111       1111111111111111      2
     3333333333333333333333333333       222222    111111     11111111111111111     2
30 98.0  2.0  3.0  3.0  8.0  8.0 -0.0 83.0 60.0 34.0 19.0 23.0 31.0 36.0 39.0 48.0 70.0
           33333333333333333333333333     2222    1111111111111111111           222
             333333333333333333333333333   2222    11111111111111111          2222
29 83.0 95.0 -0.0 -0.0 10.0 22.0 28.0 12.0 81.0 53.0 29.0 19.0 28.0 43.0 51.0 60.0 85.0
     2                333333333          3333   2222    1111111111111111        22222
     22               333333333           333    222    1111111111111111     222222
28 70.0 87.0 95.0 -0.0  7.0 23.0 40.0 20.0 90.0 61.0 38.0 23.0 24.0 43.0 60.0 78.0 10.0
     2222           333333333333         3333    222     11111111111111    22222    33
      2222         3333333333333333   333333   2222    111111111111111     2222   333
27 53.0 78.0 98.0  5.0  3.0  8.0 17.0  8.0 81.0 53.0 34.0 26.0 28.0 45.0 73.0  3.0 40.0
     CONTOUR PROGRAM TOOK    0.23 SECONDS.
```

Fig. D.19 Contour map showing same region as Fig. D.18, using smaller grid squares with linear interpolation.

Fig. D.20 Contour map using method of printing characters at positions through which contours pass.



MAXIMUM VALUE IS -1.1
MINIMUM VALUE IS -9.7
CONTOUR INTERVAL IS .5

Fig. D.21 Contour map of a computed function.

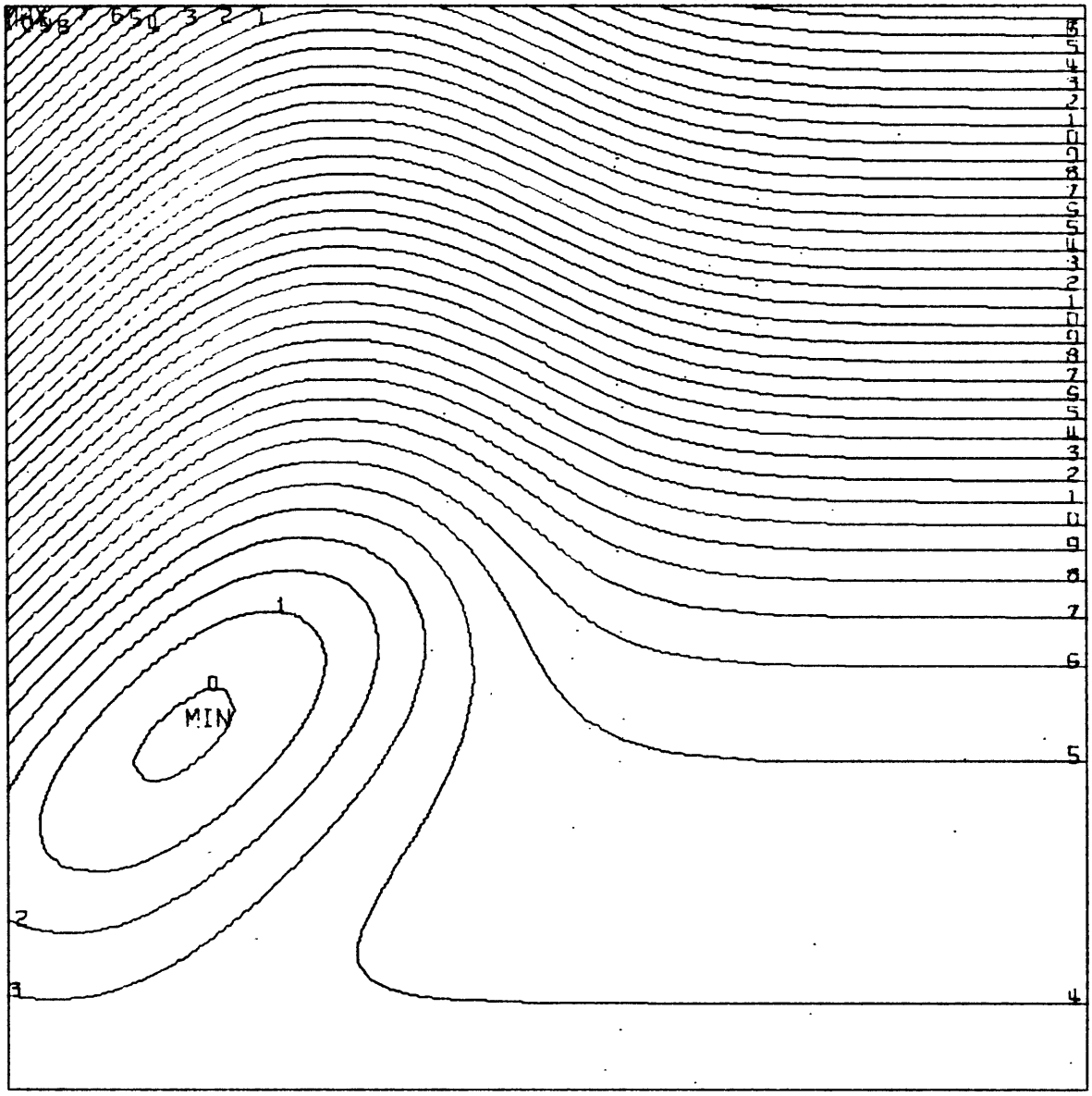


Fig. D.22 Contour map of a computed function.

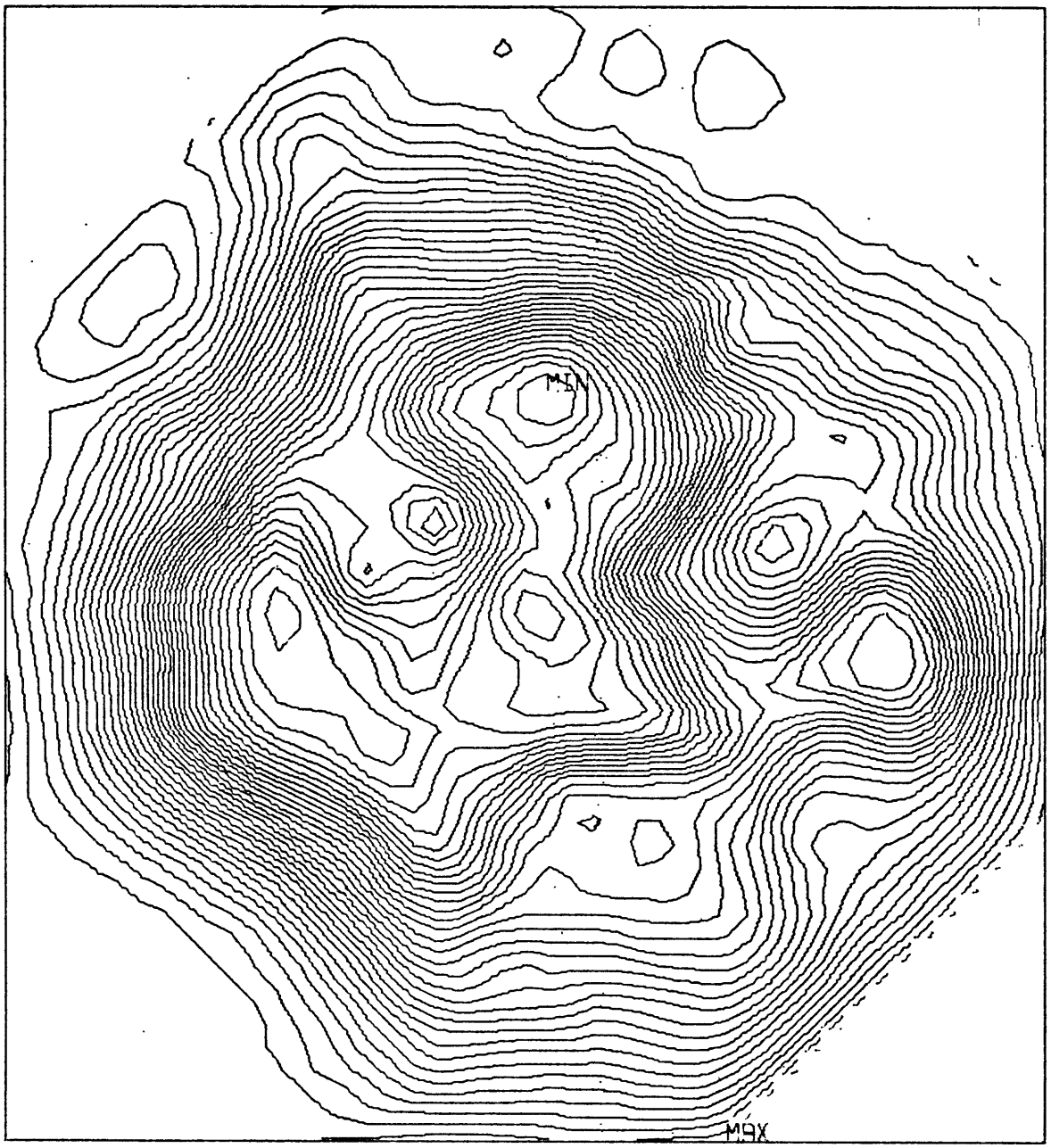


Fig. D.23 Contour map of 500 mb. geopotential heights. (Meteorological data at about 18000 feet of altitude).

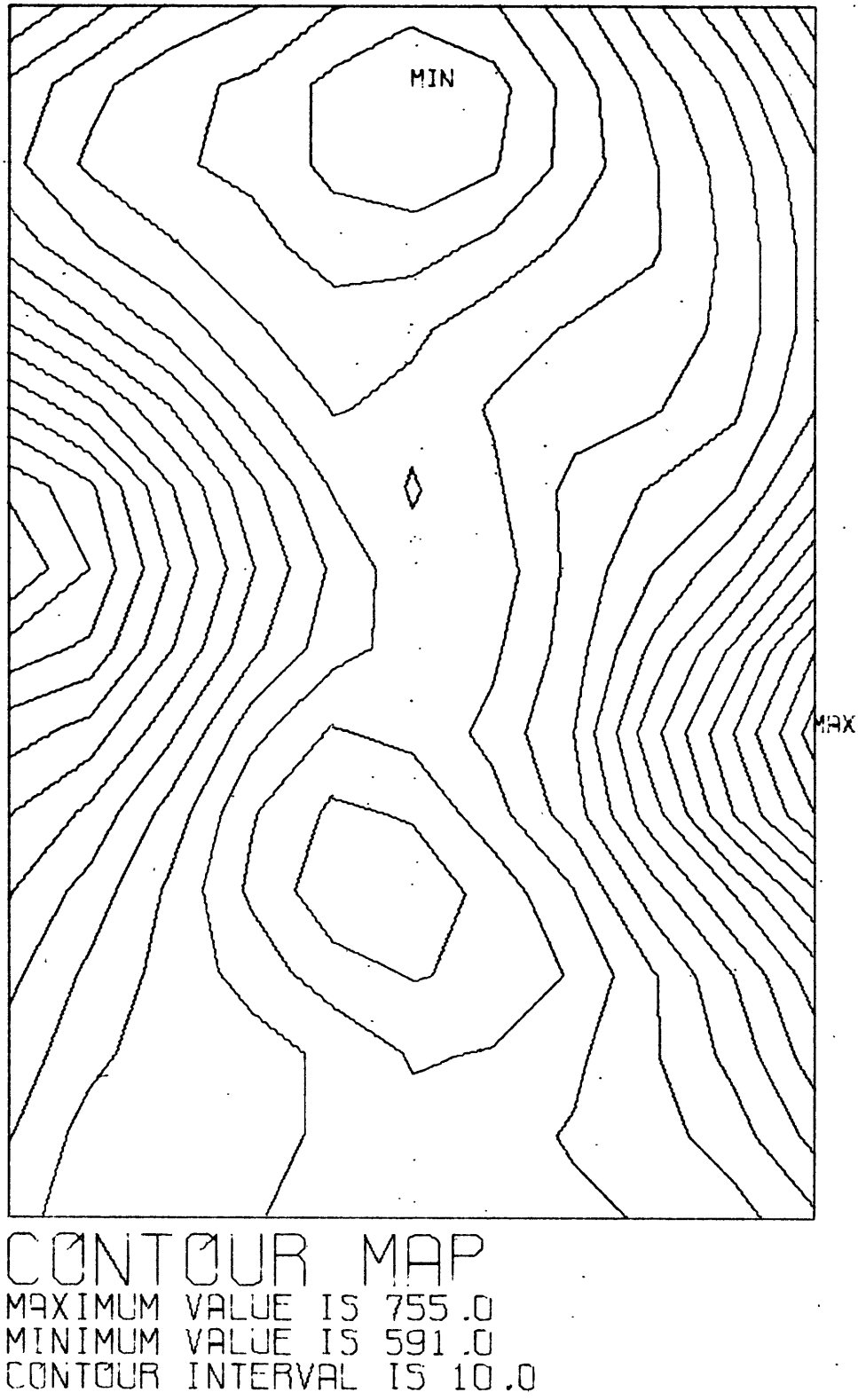


Fig. D.24 Contour map of a small section from the array contoured in Fig. D.23.

APPENDIX E

WORLD MAP PLOTTING AND PROJECTIONS

E.1 Introduction to Mapping

The problem of producing maps of the geographic features (such as continents, lakes, etc.) on the face of the earth has been of little concern to the fluid scientist in the past. It has usually been solved by printing maps of the areas which interest meteorologists, where data is taken, or where particular numerical prediction grids are located. However, as meteorologists and oceanographers have become interested in new regions (such as the southern hemisphere or Pacific equatorial islands) and have tried to work with special grids in these areas, it has become a necessity to create or acquire new types of specialized maps. Furthermore, as numerical modeling of terrestrial fluids progresses and takes more account of geographic and topographic features, it becomes important to have objective descriptions of those features stored within the system of programs the fluid scientist uses. Finally, the fact that the computer has the ability to produce contour maps in an immense variety of projections, magnifications and shapes, requires us to achieve the same flexibility in the production of geographic outlines for physical position referencing, if we are to be able to utilize the complete range of variation open to us. All of this points to

the fact that future systems for the fluid sciences must include quite complete information about the geography and topography of the earth.

E.2 The Data

As an initial attack on the problem, beginning the formation of a data foundation, a compactly encoded map of the geographic outlines of earth's continents, islands and bodies of water has been produced. This map has been encoded from maps received from TRW Systems in Manhattan Beach, California and the National Center for Atmospheric Research in Boulder, Colorado. These maps, as received, contained great quantities of detailed information, but were encoded in such a way that they required large numbers of cards, and large blocks of computer storage. The encoding developed replaces each latitude-longitude pair by a single nine digit decimal number representing both angles in degrees and hundredths. Hundredths of degrees appeared sufficient precision since that represents a maximum error in placement of any point due to truncation of about 1/2 mile. The encoding used has the advantage that it can be decoded easily by inspection. The first four digits of the nine digit number represent latitudes from 0.00 to 90.00 degrees. Whether the point is in the northern or southern hemisphere is represented by adding 500 degrees to the longitude for negative (southern hemisphere) latitudes. The last five digits of the nine digit number represent longitudes from 0.00 to 360.00 degrees. In the southern hemisphere the range is

500.00 to 860.00. Since the largest number possible is 900086000, this value can be easily stored in 30 bits as a binary integer. The end of a sequence of points which describe an outline is marked by a value of zero.

The 8853 points in the map require 1107 cards when punched on cards as nine digit decimal numbers, 8 to the card. When the binary form of the numbers is punched in A4 format, 18 to the card, only 492 cards are required. This latter form, though very compact, cannot be easily decoded or corrected by hand. The nine digit decimal form is the most compact form compatible with easy modification and addition by human beings using card punches. It is hoped that the future will bring wide dissemination of this map to fluid scientists, and greater completeness of the map as users add information to areas of special interest.

E.3 The Program

Once encoding and compact formatting of the map had been completed, it was necessary to create a program which could plot the map in different projections and from different points of view. This was first accomplished on the ARDS II (Stotz 1967) graphical display console at project MAC under an author-run project called DISHPAN (Gammill 1968a). That program is still serving as a demonstration program for the ARDS II, and is quite popular because of the complex and recognizable result produced, as well as the

flexibility with which point-of-view, magnification, projection, precision and blanking of hidden lines can be controlled. A man-machine interaction environment makes the flexibility (or lack of it) of a program much more apparent than does batch processing. Since the completion of the DISHPAN project, the program has been recoded in FORTRAN IV to operate the CALCOMP plotter from tapes produced on the IBM 360 model 65. In the following pages the projection and point of view transformation equations are given.

E.4 Projections

There are two basic approaches to projecting the surface of a sphere onto a flat map. These could be termed projection onto a cylinder and projection onto a plane. Projections onto cylinders have the advantage of presenting all (or nearly all, in the case of the mercator projection) of the surface of the earth on a single map, but the disadvantage of horrendously distorting shapes and/or distances at the ends of the cylinder. If, as is usually the case, the axis of the cylinder is the same as the axis of the earth, this means that cylindrically projected maps faithfully represent the equatorial regions but distort the polar regions. Projections onto planes have the advantage of being somewhat more faithful representations of the appearance of the earth over the region they are able to cover, but that region usually is no more than a single hemisphere. Because of this restricted scope, projections onto planes require the ability

to position the plane over any desired point on the surface of the earth. This transformation of "point of view" allows the relatively faithful presentation of any region smaller than a hemisphere. Point of view transformation for cylindrical projections is not normally done, and will not be examined here. The major use of such a transformation appears to be the calibration of strip photographs from satellites.

The cylindrical projections, because of the lack of point-of-view transformation, tend to be quite simple. The simplest possible projection is the cylindrical equi-spaced which maps the longitude θ and latitude ϕ (scaled to an appropriate range) directly onto the x and y axes of the map plane. The equations are:

$$x = k(\theta - \theta_0)$$

$$y = k\phi$$

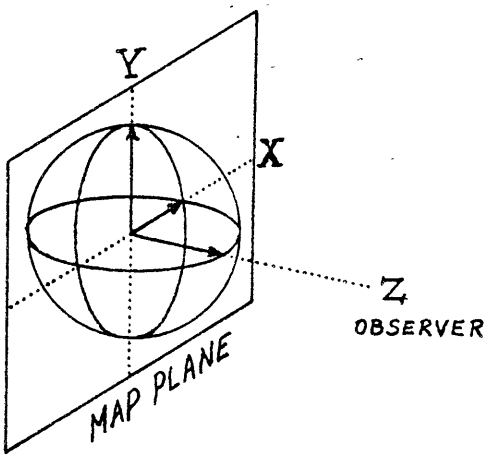
The constant k is a scaling factor and θ_0 is the longitude which is placed at the center of the map. This projection preserves true distance along meridians. Another widely used cylindrical projection is the mercator. This projection produces lines of projection between sphere and cylinder which pass through the center of the sphere. This means that points near the poles will be projected very far out on the cylinder, so that polar regions cannot be shown if our map is to be bounded at top and bottom. The equations are:

$$x = \frac{k}{b}(\theta - \theta_0)$$

$$y = b \tan\phi$$

The constants k and b are scaling factors. The mercator projection preserves the shapes of objects, but distorts all distances and sizes.

The projections onto planes require us, as was explained earlier, to include a point of view transformation. To accomplish this transformation in the simplest possible manner requires us to go from our latitude-longitude coordinate system to a cartesian axis system with the origin at the earth's center, the Y axis pointing toward the north pole, the Z axis pointing at the intersection of the equator and the Greenwich meridian, and the X axis pointing at the equator and longitude 90^{o} East. In the unrotated case this axis system coincides with the observers axis system (x, y and z) where the observer is on the positive z axis.



$$X = \sin\theta \cos\phi$$
$$Y = \sin\phi$$
$$Z = \cos\theta \cos\phi$$

Fig. E.1 The axis system.

In this untransformed case if an orthographic projection centered over the intersection of the equator and the Greenwich meridian, is desired, it is only necessary to use the X and Y axes as the map

axes, x and y. However, we need the ability to center our map over any longitude θ and latitude ϕ desired, and to rotate the longitude circle θ_o away from the map's vertical axis y. To accomplish this requires us to rotate the X, Y, Z axis system successively around the y, x and z axes by the angles θ_o , ϕ_o and α .

A rotation of θ_o around the y (map vertical) axis produces:

$$x_1 = X \cos\theta_o - Z_1 \sin\theta_o$$
$$y_1 = Y$$
$$z_1 = Z_1 \cos\theta_o + X \sin\theta_o$$

A rotation of ϕ_o around the x (map horizontal) axis produces:

$$x_2 = x_1$$
$$y_2 = y_1 \cos\phi_o - z_1 \sin\phi_o$$
$$z_2 = z_1 \cos\phi_o + y_1 \sin\phi_o$$

A rotation of α around the z (observer) axis produces:

$$x = x_2 \cos\alpha - y_2 \sin\alpha$$
$$y = y_2 \cos\alpha + x_2 \sin\alpha$$
$$z = z_2$$

Thus:
$$x = R_{11} X + R_{12} Y + R_{13} Z_1$$
$$y = R_{21} X + R_{22} Y + R_{23} Z_1$$
$$z = R_{31} X + R_{32} Y + R_{33} Z_1$$

where:

$$R_{11} = \cos\theta_o \cos\alpha + \sin\theta_o \sin\alpha \sin\phi_o$$
$$R_{12} = -\cos\phi_o \sin\alpha$$
$$R_{13} = \cos\theta_o \sin\alpha \sin\phi_o - \sin\theta_o \cos\alpha$$
$$R_{21} = \cos\theta_o \sin\alpha - \sin\theta_o \cos\alpha \sin\phi_o$$
$$R_{22} = \cos\phi_o \cos\alpha$$
$$R_{23} = -\cos\theta_o \cos\alpha \sin\phi_o - \sin\theta_o \sin\alpha$$
$$R_{31} = \sin\theta_o \cos\phi_o$$
$$R_{32} = \sin\phi_o$$
$$R_{33} = \cos\theta_o \cos\phi_o$$

At this point we can produce an orthographic projection for any point-of-view by specifying values for θ_o, ϕ_o and α, computing the rotation matrix R, and then using it to rotate the X, Y, Z axes to x, y and z, whereupon x and y are used as map coordinates. The z component is also important, for if we do not wish to see outlines from the backside of the globe we must be careful not to plot data points for which z is less than zero.

The orthographic projection produces a view of the globe from infinite distance, with all lines of sight parallel. That is a special case of the perspective projection, where the globe can be viewed from various distances. To correct for perspective simply requires corrections to the plotted x and y coordinates for the relative distance from the observer's position.

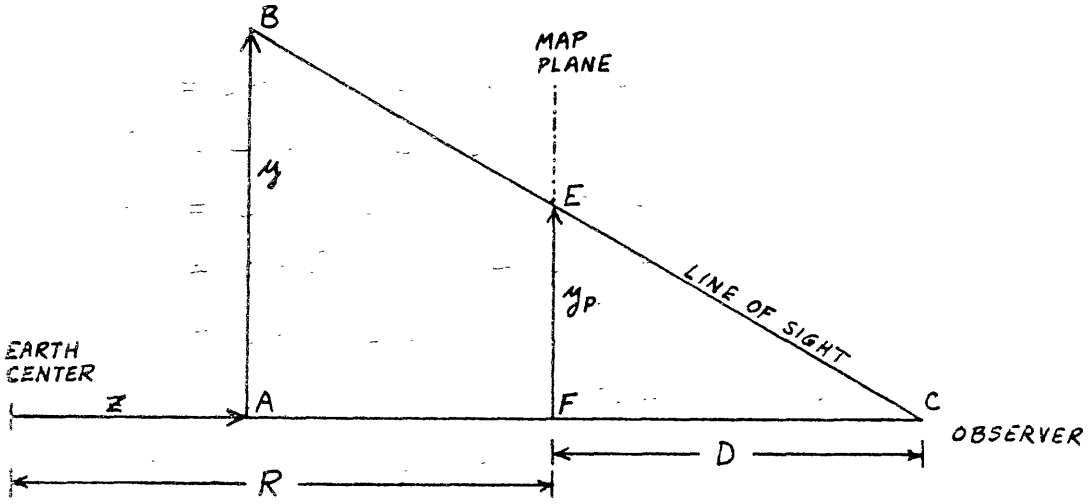


Fig. E.2 Diagram for perspective transformation.

By congruence of triangles ABC and FEC in Figure E.2:

$$\frac{y}{R+D-z} = \frac{y_p}{D}$$

$$\frac{x}{R+D-z} = \frac{x_p}{D}$$

$$x_p = \frac{D}{R+D-z}\,x$$

$$y_p = \frac{D}{R+D-z}\,y$$

A further problem produced when the perspective transformation is added is that the prevention of backside images can no longer be handled by not plotting points where $Z < 0$. Now we must not plot points which are beyond the point of tangency of line of sight and the surface of the globe. The point of tangency is normally called the earth's limb, so the equation to be satisfied can be termed the limb condition.

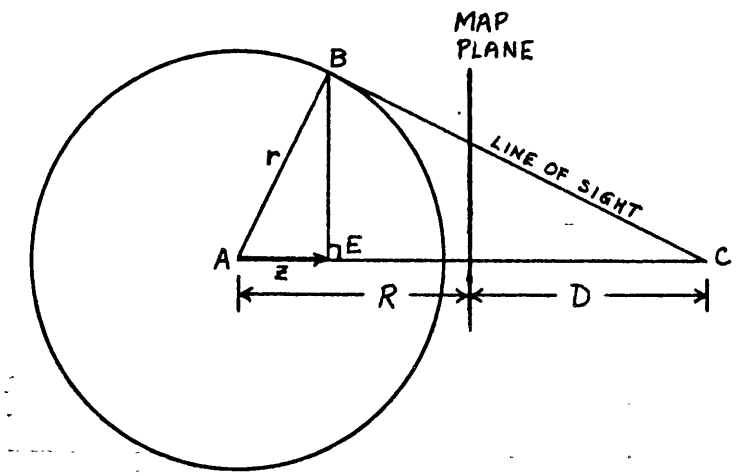


Fig. E.3 Diagram of limb condition for perspective views.

Regarding Figure E.3, since the line of sight CB forms a right angle with AB, and AE is perpendicular to EB, we can state congruence of AEB and ABC. Thus the limb condition is stated by:

$$\frac{z}{r} = \frac{r}{R+D}$$

$$z = \frac{r^2}{R+D}$$

Thus, to be visible on our map, a data point should satisfy the inequality:

$$\boxed{z \geq \frac{r^2}{R+D}}$$

These equations can be seen to be consistent since as D becomes very large, the perspective correction approaches one and the limb condition approaches zero, which gives the orthographic projection and limb condition.

Another projection on a plane is the equidistant projection, which preserves true distance from the center of the map. Thus, the axes x and y must be transformed back into angular displacements, which are the map axes.

$$x_E = \frac{kx}{\sqrt{r^2 - z^2}} \cos^{-1}\left(\frac{z}{r}\right)$$

$$y_E = \frac{ky}{\sqrt{r^2 - z^2}} \cos^{-1}\left(\frac{z}{r}\right)$$

with $Z \geq 0$

Here k is a scaling factor and r is the radius of the earth in map units.

The final projection on a plane that we will examine is the stereographic projection. This projection maps points from the sphere onto a tangent plane from a projection point on the opposite side of the sphere.

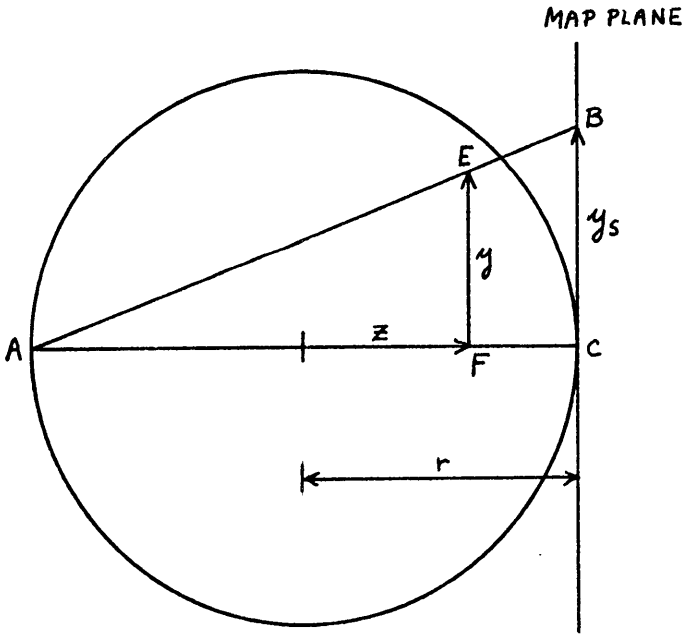


Fig. E.4 Diagram of stereographic projection

By congruence of triangles ABC and AEF in Figure E.4:

$$\frac{y_s}{2r} = \frac{y}{r+z}$$

$$\frac{x_s}{2r} = \frac{x}{r+z}$$

$$\boxed{\begin{aligned} x_s &= \frac{2r}{r+z}\, x \\ y_s &= \frac{2r}{r+z}\, y \end{aligned}} \qquad \text{with} \quad Z \geq 0$$

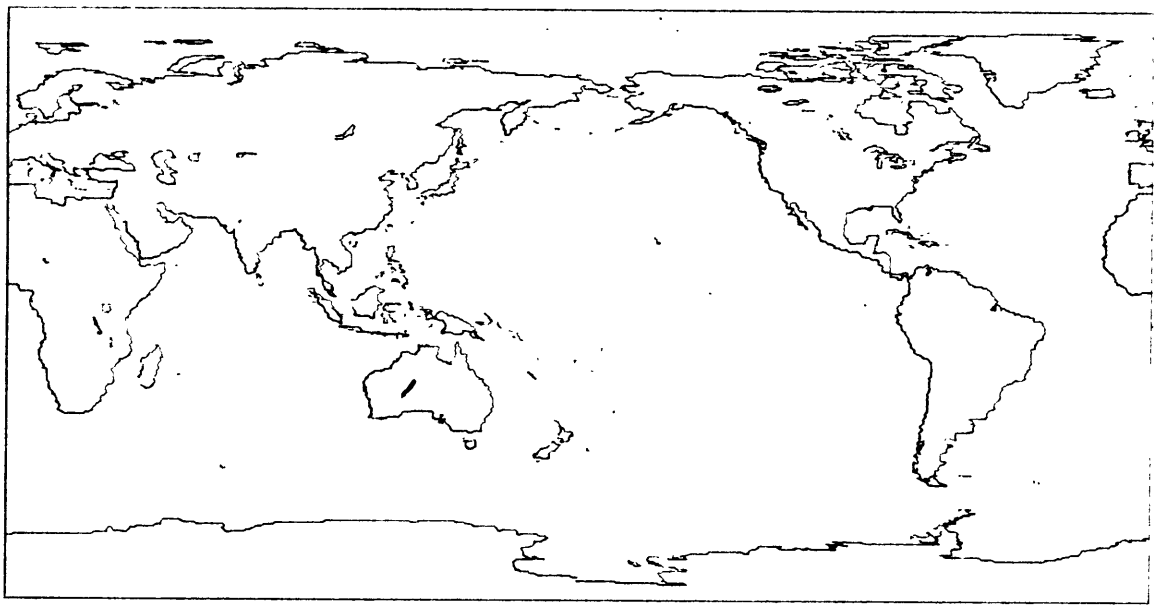


Fig. E.5 Cylindrical equispaced projection of the whole earth.

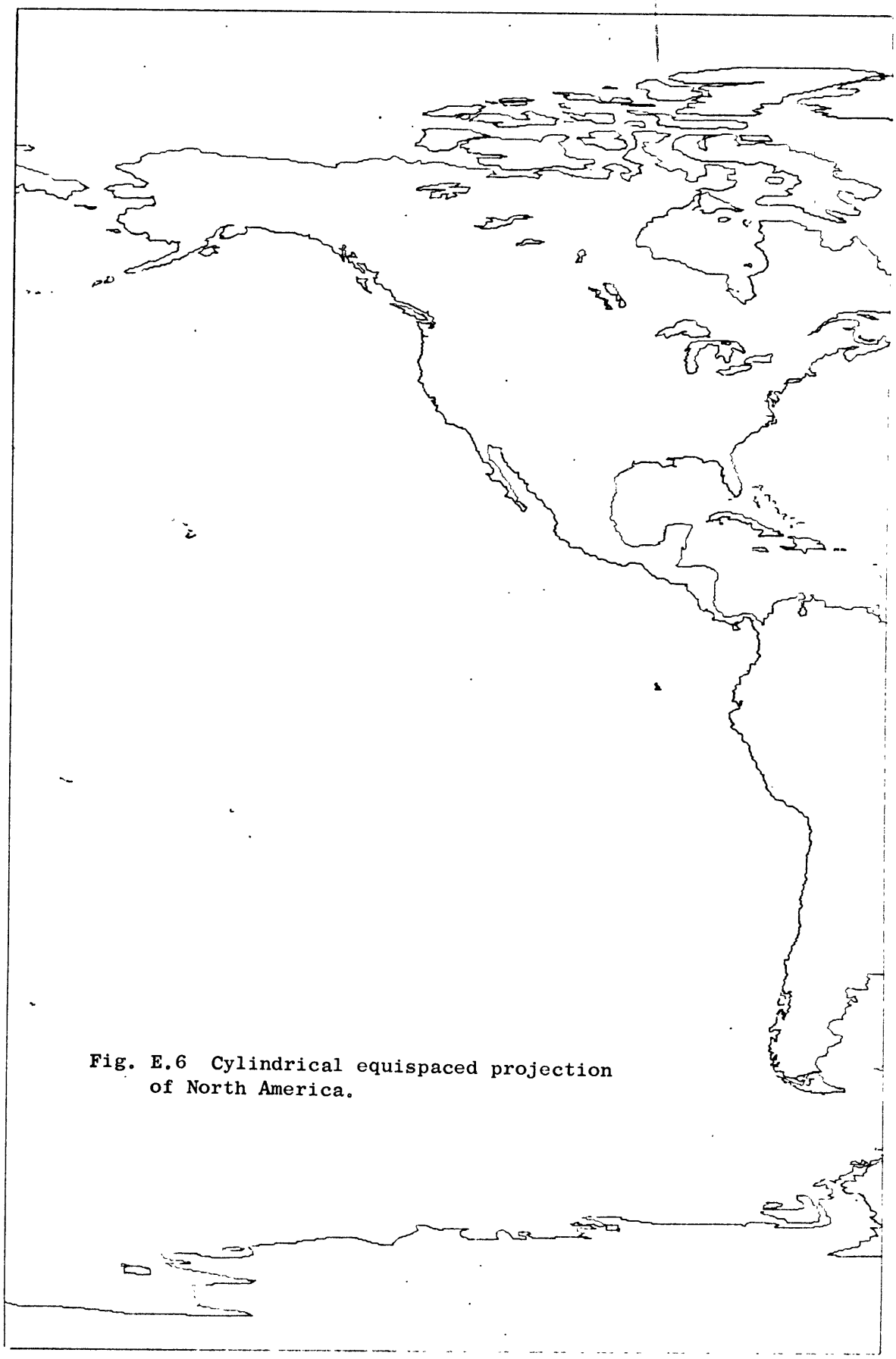


Fig. E.6 Cylindrical equispaced projection of North America.

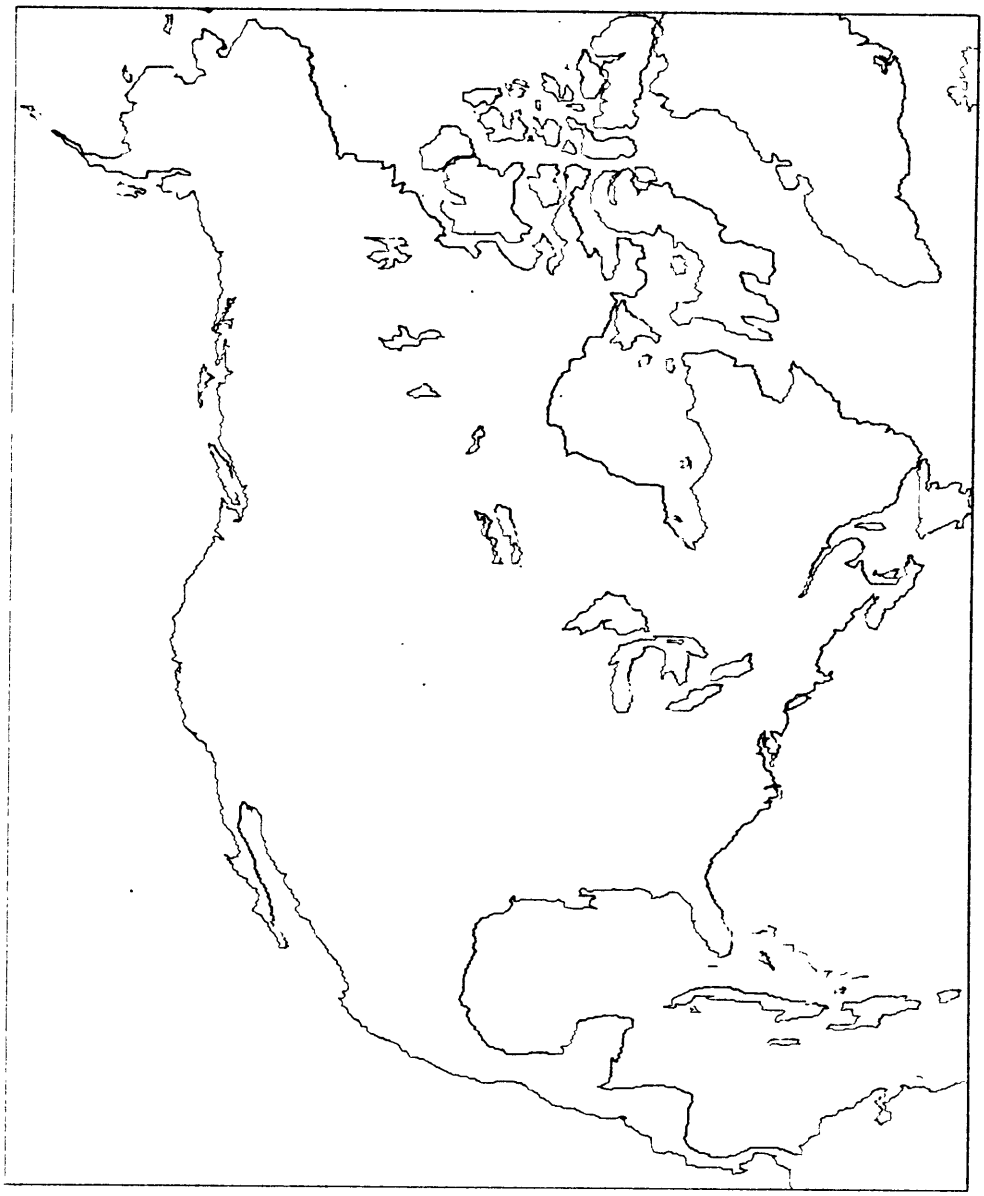


Fig. E.7 Orthographic projection of North America.

Fig. E.8 Orthographic projection with point of view at the North Pole.

Fig. E.9 Stereographic projection with point of view at the North Pole.

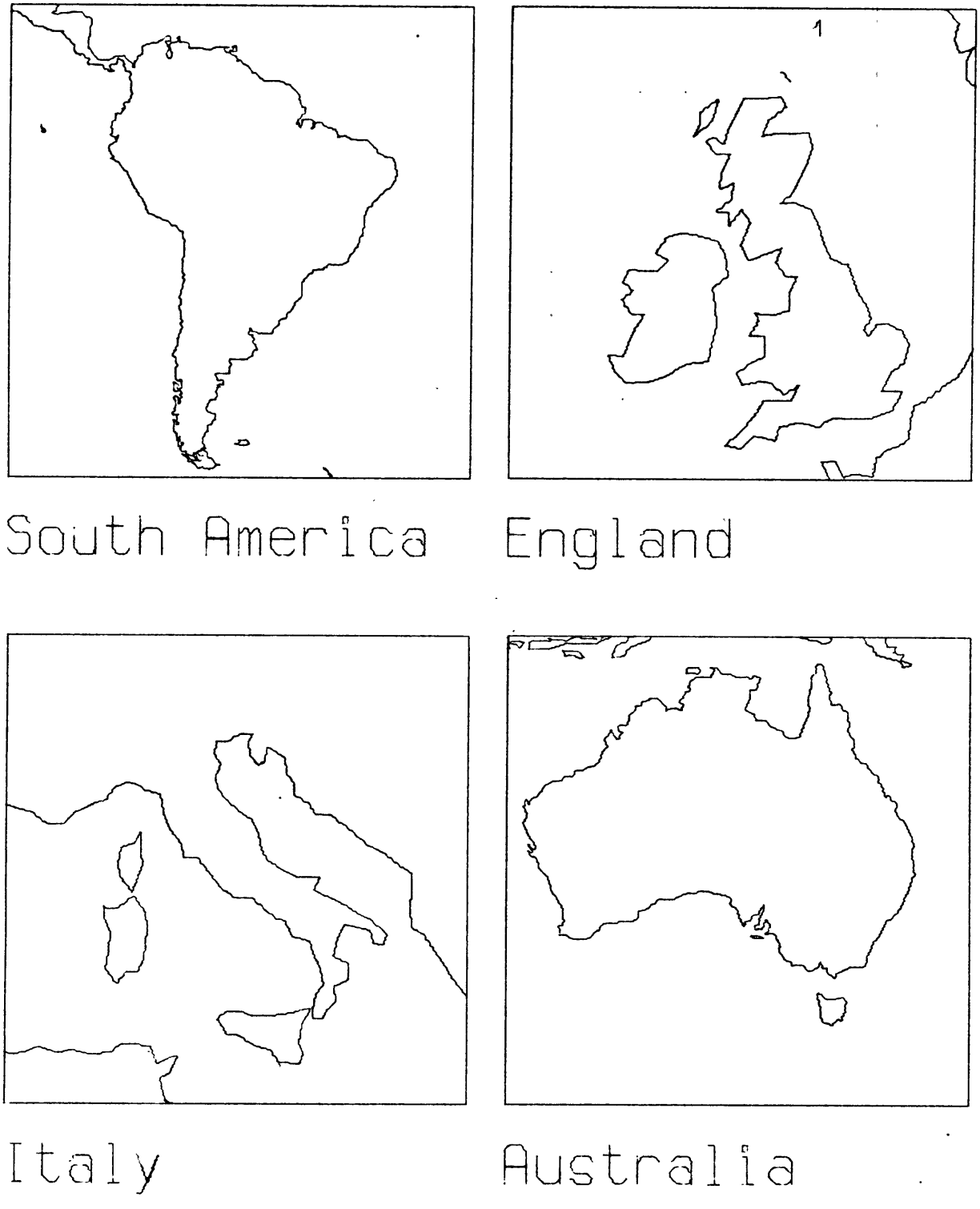


Fig. E.10 Orthographic maps of countries.

APPENDIX F

F.1 Special Library Programs

A number of service programs are contained in the library, and can be CALLed by the evaluator, but are not actually named members of that library. These programs are contained as entry points in two special load modules that are members of the library. Those members are named CALFACE and FIOFACE, indicating that the first contains interface routines for the Calcomp plotter and the second contains interface routines for managing FORTRAN input-output.

The names of the service programs available in this manner are:

(a) Calcomp - NEWPLT,ENDPLT,PLOT1,LRCASE,SYMBL5,
NUMBR1,GRAPH,PICTUR,SCLGPH

(b) Fortran - ERRSET

(d) DUMP

(d) CLOCK

F.2 Decompositon and Composition Operators of the STRAN Language

A decomposition operator string is made up of the following operators separated by plus signs:

(a) $ - matches any string including the null string.

(b) literal - character string surrounded by single quote marks. Matches only itself.

(c) storage name - this storage location must contain a sequence of literals separated by single quotes and terminated by two right parentheses. The first of these literals to produce a satisfactory match is used. An example of the contents of the storage location is: ' THE ' A ' AN '))

(d) $n - dollar sign followed by an integer, matches the first n characters of the remaining string.

(e) $'character string' - dollar sign followed by a literal, matches as many repetitions of the literal as can be found in consecutive order. This includes matching the null string.

(f) @n - at sign followed by an integer, matches everything up to but not including the n-th character. If the matching process has already passed character n, then this matches the null string, and matching continues from character n.

(g) .n - period followed by an integer, causes the contents of pseudo-register n to be inserted (at execution time) in the operator string, for this operator.

If a literal occurs in the left or rightmost position of a decomposition operator string, the left or rightmost portion of the string to be matched must duplicate the literal exactly. Dollar signs allow the matching process to 'float', and a literal preceded by a dollar sign will match the leftmost occurrence of itself. The elements of the match are placed into successive pseudo-registers where they remain until obliterated by the left side of some later rule.

A composition operator string is made up of the following operators separated by plus signs:

(a) integer from 1 to 10 - these refer to the ten pseudo-registers.

(b) literal - a character string surrounded by single quote marks.

(c) @n - at sign followed by an integer, causes composition to continue at column n, either by adding blanks or truncating.

(d) %Ln,m - percent sign followed by an L followed by two integers separated by a comma, causes the left-most n characters of the string in pseudo register m to be concatenated to the result. If there are less than n characters, the string is padded on the right with blanks.

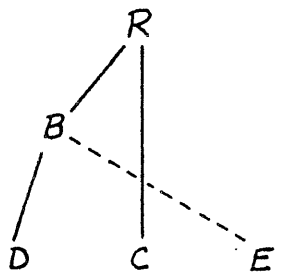
(e) %Rn,m - same as (d) except right-most n characters are used. If not enough characters are available, padding with blanks is on the left.

F.3 Grammar to Generate Statements for COMS

On a following page we present the grammar used by the COMS presented in Sections 8.2 and 8.3. This grammar is a context free deterministic (discontinuous) phrase structure grammar, using rules of the type described by Yngve (1960). There are five types of rules.

1. (A= <abc>) symbol A generates the character string "abc"
2. (A=B) symbol A generates the symbol B
3. (A*3) symbol A has three alternative subsymbols, A:1, A:2 and A:3, which it may generate.
4. (R=B+C) symbol R generates the symbol B followed by the symbol C.
5. (B=D...E) symbol B generates the symbol D, followed by the symbol that follows B, followed by C.

The fifth type of rule, invented by Yngve, will surely cause some confusion. Perhaps the operation of this type of rule is most easily clarified by saying that if a generation is started with the symbol R, the above rules will generate the sequence of symbols DCE. A graph of this generation is:



Two extra facilities have been included in this grammar, to allow character strings to be inserted from the STRAN storage or associative memory. A symbol beginning with &, for example &ARG, will cause the contents of STRAN storage location ARG to be inserted. Likewise, a symbol beginning with period, for example .PROGRAMMER_NUMBER, will cause insertion of the string retrieved by a FIND request to the associative memory of the form (PROGRAMMER_NUMBER,$).

```
STRAN INTERPRETER ON MAY 24,1969 AT 13:34:13.690 PAGE  9


BEGIN INTERPRETING RULES.


        FOLLOWING INPUT STORED AS GRAMMAR RULES.
INPUT...(%CALP=%CALL+&PROGNAM)(%CALL=<CALL >)(%CWP=%LH...%RH)(%LH=%CALP+%LP)
INPUT...(%RH=%RP+%SC)(%CALCON=%CWP+%ALST)(%ALST=%AL1+%AL2)(%AL1=&ARGS)
INPUT...(%AL2*?)(%AL2:1=<,IDIM,JDIM>)(%AL2:2=<,47,51,0,0,-1>)(%COMA=<,>)(%SC=<;>)
INPUT...(%SQT=<'>)(%LP=<(>)(%RP=<)>)
INPUT...(%SIMPCAL=%CALP+%SC)(%CALOPN=%CWP+%NALS)(%NALS=%NAPN+%NAL1)(%NAL1=%COMA+%NAL2)
INPUT...(%NAL2=%NARN+%NAL3)(%NAL3=<,'VELLUM','BLACK'>)(%NAPN=%QUOTE+.PROBLEM_NUMBER)
INPUT...(%NARN=%QUOTE+.PROGRAMMER_NUMBER)(%QUOTE=%SQT...%SQT)
INPUT...(%READ=%CWP+%RDLST)(%RDLST=%STUFF+&ARGS)(%STUFF=%ST1A...%ST2)
INPUT...(%ST1A=%ST1C+%COMA)(%ST1C=%ST1D+%ST1E)(%ST1D=<5,>)(%ST1E=%QUOTE+%ST1F)
INPUT...(%ST1F=%ST1G+%RP)(%ST1G=<(24F3.C/23F3.0>)(%ST2*2)(%ST2:1=<,1,IDIM*JDIM>)
INPUT...(%ST2:2=<,1,2397>)(%CALCCG=%CALCON+%GLOBE)
INPUT...(%GLOBE=%GLCBP+%MOVE)(%MCVE=%CPL1+%RH)(%CPL1=<CALL PLCT1(14.0,0,-3>)
INPUT...(%GLOBP=%GLB+%RH)(%GLB=%GLB1+%GLB2)(%GLB1=<CALL GLOBE(2,90.,280.,0,>)
INPUT...(%GLB2=<4.070136,0,2.,6.,8.52>)
INPUT...(%CALCOM=%CALCON+%MOVE)
INPUT...)))))
        END OF GRAMMAR.
```

Grammar to generate statements for COMS.

F.4 STRAN Rule Set for a Sophisticated COMS

The following rule set implements the COMS demonstrated in Sections 8.2 and 8.3. EVALUATE is the name of the initial rule of this set. That rule calls each of the five rule sets which implement the five phases of execution of this COMS, described in Section 8.2. Each of the basic rule sets, which make up this total collection, are separated from preceding rules by a blank card, and major subsets are headed by comment cards describing the function of that subset.

The first page of the following material contains a listing of the OS360 Job Control Language (JCL) input cards which are necessary in order to run COMS. This is presented for the information of experienced users of OS360, and will not be explained here. Following the input //SYSIN DD *, is a card which specifies to the interpreter (ALLOC.ALLOCAT='1'B) that the associative memory is to be used, that the number of lines to be printed on an output page is 33, and that those pages are to be printed in report format. More complete explanation of these capabilities will have to await the writing of a manual. These control cards are all that is necessary in order to use the interpreter and library. Once these cards are read, the following STRAN rules may be given, or rules may be collected from secondary storage (DDNAME=SYSDATA) by command.

```
//J51083   JOB (M6350,2024,3,1999,900,SRI-0),'GAMMILL X5932',MSGLEVEL=1
//JOBLIB DD DSNAME=USERFILE.M6350.2024.LOAD.COMSLIB,DISP=OLD
//INTERP EXEC  PGM=INTRPRTR
//FT06F001 DD  SYSOUT=A,DCB=(RECFM=VBA,LRECL=136,BLKSIZE=769)
//FT07F001 DD  UNIT=(SYSCP,,DEFER),DISP=(MOD,PASS),                    X
//             DCB=(RECFM=F,LRECL=80,BLKSIZE=80)
//FT10F001 DD  DSNAME=USERFILE.M5270.2024.DATA.MAP,DISP=OLD
//SYSPRINT DD  SYSOUT=A
//SYSDATA  DD DSNAME=USERFILE.M5270.2024.DATA.COMS,DISP=OLD
//PL1DUMP  DD  SYSOUT=A
//FT05F001 DD  DDNAME=SYSIN
//SYSIN DD *


REPORT='1'B, LINLIM=33, ALLOC,ALLOCAT='1'B;
```

BEGIN READING RULES.

```
INPUT...(EVALUATE(BEGPRT,PARSE,DEDUCE,INITPRT,INITIALIZE,CMPRINT,COMPUTE,CLOSECUT))
INPUT...
INPUT...(BEGPRT(=*OUT|'FOLLOWING INPUT DECLARATIONS PROCESSED.'|*OUT|' '|)END)
INPUT...
INPUT...# PARSE ACCEPTS ENGLISH INPUT, TRANSLATES IT TO STL AND STORES RESULTS IN
INPUT...#  THE ASSOCIATIVE MEMORY. THIS RULE SET CALLS ASSERT.
INPUT...(PARSE(READ,BEGIN,STORE,PTEST))
INPUT...(PTEST(INPUT|$+')'+$|)PARSE,END)
INPUT...(BEGIN(INPUT|$+'.  '+$|=*OUTPUT|' '|*SENTENCE|1+'.'|INPUT|3|)BEGUN,B1)
INPUT...(BEGUN(INPUT|$+'*'+$+'.'+$|=*OUTPUT|' '|*SENTENCE|2+3+4|INPUT|5|)END,DBL)
INPUT...(B1(SENTENCE|' '+$|=SENTENCE|2|)DBL,B1)
INPUT...(DBL(SENTENCE|$+'  '+$' '+$|=SENTENCE|1+' '+4|)RULE1,DBL)
INPUT...(RULE1(SENTENCE|$+' IS '+$+'.'|=NP|1+' '|VP|' '+3|)ACTV,VPD1)
INPUT...(VPD1(VP|DETS+$|=VP|' '+2|VPTYP|'NOUN,'|)VPD3,RULE2)
INPUT...(DETS(' A ' AN '))
INPUT...(VPD3(VP|' THE '+$|=VP|' '+2|VPTYP|'FUNCTION,'|)VPD4,RULE2)
INPUT...(VPD4(=VPTYP|'ADJECTIVE,'|)RULE2)
INPUT...(RULE2(NP|'*HE '+$|SUBJ|$|=OUT|'MALE,'+3|TMP|'S2,PREPS))'|)RULE3,TMP)
INPUT...(RULE3(NP|'*SHE '+$|SUBJ|$|=OUT|'FEMALE,'+3|TMP|'S2,PREPS))'|)RULE4,TMP)
INPUT...(RULE4(NP|'*IT '+$|SUBJ|$|=OUT|'NEUTER,'+3|TMP|'S2,PREPS))'|)RULE5,TMP)
INPUT...(RULE5(NP|D1+$+' '|=OUT|'NOUN,'+2|SUBJ|2|TMP|'S2,RSUBS))'|)RULE6,TMP)
INPUT...(D1('*ANY '*EVERY '))
INPUT...(RULE6(NP|D2+$+' '|=OUT|'ADJECTIVE,'+2|SUBJ|2|TMP|'S2,RSUBS))'|)RULE7,TMP)
INPUT...(D2('*ANYTHING '*EVERYTHING '))
INPUT...(RULE7(NP|'*NO '+$+' '|=OUT|'NOUN,'+2|SUBJ|2|TMP|'S2,RDISJ))'|)RULE8,TMP)
INPUT...(RULE8(NP|'*NOTHING '+$+' '|=OUT|'ADJECTIVE,'+2|SUBJ|2|)RULE9,RULE8A)
INPUT...(RULE8A(=TMP|'S2,RDISJ))'|)TMP)
INPUT...(RULE9(NP|$+PREF+$+' '|=SUBJ|3|)GSUBJ,PREPS)
INPUT...(GSUBJ(NP|$+' '|=SUBJ|1|)PREPS)
INPUT...(RSUBS(SUBJ|$|VP|' '+$|=RSS|'(SUBSET OF,'+1+','+3+')'|)RVPT)
```

```
INPUT...(RDISJ(SUBJ|$|VP|' '+$|=RSS|'(DISJOINT FROM,'+1+','+3+')'|)RVPT)
INPUT...(PREPS(VP|$+PREPOSITNS+$|=RELTN|1+2|OBJ|3|)TEST,3T3)
INPUT...(PREPOSITNS(' OF ' IN ' TO ' THAN ' FROM ' AT ' FOR ' BETWEEN ' ON '))
INPUT...(2TUP(VP|' '+$+','+$|=VP|4|)2T1,2TA)
INPUT...(2T1(VP|' AND '+$|=VP|' '+2|)2T2,2T3)
INPUT...(2T2(VP|' '+$+' '+$|=VP|' '+4|)2T3,2TA)
INPUT...(2T3(VP|' '+$|SUBJ|$|=RSS|'('+2+','+3+')'|)RVPT)
INPUT...(2TA(SUBJ|$|=OUT|'ADJECTIVE,'+2|RSS|2+','+1|TMP|'S2,2TB,S2,2TUP))'|)TMP)
INPUT...(2TB(RSS|$|=OUT|1|)END)
INPUT...(SPSHL('S OF 'S IN 'S TO 'S FROM 'S AT 'S FOR 'S ON '))
INPUT...(ACTV(SENTENCE|$+' '+$+SPSHL+$+'.'|=NP|1+2|OUTPUT|3|VP|4+5|)KNRLTN,A1)
INPUT...(A1(NP|$|OUTPUT|$+' '+$|=NP|1+2+' '|OUTPUT|4|)A2,A1)
INPUT...(A2(OUTPUT|$|VP|$|=VP|' '+1+2|VPTYP|'VERB PHRASE,'|)RULE2)
INPUT...(KNRLTN(&F1|'RELATION'+$|=VPTYP|'VERB ROOT'|)UNABLE,KNR1)
INPUT...(KNR1(&A1|1|=OUTPUT|'' '+1+' ''|)UNABLE,KNR2)
INPUT...(KNR2(OUTPUT|$|SENTENCE|$+.1+$+'.'|=SUBJ|2|RELTN|3|OBJ|4|)KNR1,3T3)
INPUT...(PREF('THE RELATION 'THE PROPERTY 'THE WORD '))
INPUT...(ALST(' AND THE PROPERTY ' AND THE RELATION '))
INPUT...(3T3(OBJ|PREF+$|=OBJ|2|)3T4)
INPUT...(3T4(OBJ|'ITSELF'+$|SUBJ|$|=OBJ|3+2|)3T5)
INPUT...(3T5(OBJ|$+' AND ITSELF'|SUBJ|$|RELTN|$+' '|=OBJ|1+','+3|RELTN|4+'&AND '|)3T6)
INPUT...(3T6(OBJ|$+ALST+$|RELTN|$+' '|=OBJ|1+','+3|RELTN|4+'&AND '|)3T7)
INPUT...(3T7(OBJ|'THE '+$|=OUT|'UNIQUE,'+2|OBJ|2|TMP|'S2,3T8))'|)3TA,TMP)
INPUT...(3TA(OBJ|3DETS+$|=OBJ|2|)3T8,3T9)
INPUT...(3DETS('A 'AN '))
INPUT...(3T8(OBJ|'*'+$|=OUT|'PROPER NAME,'+1+2|TMP|'S2,3TUP))'|)3T9,TMP)
INPUT...(3T9(OBJ|$+PREPOSITNS+$|=OUTPUT|2|OBJ|1+','+3|)3TUP,3TN)
INPUT...(3TN(OUTPUT|' '+$|RELTN|$+' '|=RELTN|3+'&'+2|)3TDT)
INPUT...(3TDT(OBJ|$+','+3DETS+$|=OBJ|1+2+4|)3DT1,3TUP)
INPUT...(3DT1(OBJ|$+','+'THE '+$|=OBJ|1+2+4|OUT|'UNIQUE,'+4|TMP|'S2,3TUP))'|)3TUP,TMP)
INPUT...(3TUP(VPTYP|'NOUN,'|RELTN|' '+$+' '|=OUT|'NOUN ROOT,'+3|TMP|'S2,3TF))'|)3T1,TMP)
INPUT...(3T1(VPTYP|'VERB'+$|RELTN|' '+$+' '|=OUT|'VERB ROOT,'+4|TMP|'S2,3TF))'|)3T2,TMP)
INPUT...(3T2(VPTYP|'FUNCTION,'|RELTN|' '+$+' '|=OUT|1+3|TMP|'S2,SUPER))'|)3TF,TMP)
```

```
INPUT...(SUPER(RELTN|' '+$+' '|=OUT|'NOUN ROOT,'+2|TMP|'S2,3TF))'|)3TF,TMP)
INPUT...(3TF(RELTN|' '+$+' '|SUBJ|$|OBJ|$|=RSS|'('+2+','+4+','+5+')'|)PNAM)
INPUT...(TEST(SUBJ|'*THE '+$|VP|' '+$|=SUBJ|4|VP|' '+2|VPTYP|'FUNCTION,'|)2TUP,PREPS)
INPUT...(RVPT(VPTYP|'AD'+$|VP|' '+$+' '+$|=OUT|'ADJECTIVE PHRASE,'+4+5+6|)RVP1,RVPA)
INPUT...(RVPA(=TMP|'S2,PNAM))'|)TMP)
INPUT...(RVP1(VPTYP|$|VP|' '+$|=OUT|1+3|TMP|'S2,PNAM))'|)PNAM,TMP)
INPUT...(PNAM(SUBJ|'*'+$|=OUT|'PROPER NAME,'+1+2|TMP|'S2,PRINT))'|)PRINT,TMP)
INPUT...(PRINT(RSS|$+'('+$+')'+$|=OUT|3|TMP|'S5,BEGIN))'|)BEGIN,TMP)
INPUT...(UNABLE(SENTENCE|$|=*OUTPUT|'UNABLE TO PARSE '+1|)BEGIN)
INPUT...
INPUT...#  DEDUCE PRODUCES DEDUCTIONS.
INPUT...(DEDUCE(DEDPRT,CHAIN,SUBSET))
INPUT...(DEDPRT(=*OUT|' '|*OUT|'FOLLOWING ARE DEDUCTIONS FROM DECLARATIONS.'|)END)
INPUT...#  DEDUCTION RULES. CALLED BY DEDUCE.
INPUT...(CHAIN(&F1|'CHAIN OF&AND'+$+$+$|)END,RCH1)
INPUT...(RCH1(&A1|1+2+3|)END,RCH2)
INPUT...(RCH2(&F2|2+$+$|)RCH1,RCH3)
INPUT...(RCH3(&A2|4+5|)RCH1,RCH4)
INPUT...(RCH4(&F3|3+5+$|)RCH3,RCH5)
INPUT...(RCH5(&A3|6|=&S|1+4+6|*OUT|' ('+1+','+4+','+6+')'|)RCH3,RCH5)
INPUT...(SUBSET(&F1|'SUBSET OF'+$+$|)END,SUB1)
INPUT...(SUB1(&A1|1+2|)END,SUB2)
INPUT...(SUB2(&F2|1+$|)SUB4,SUB3)
INPUT...(SUB3(&A2|3|=&S|2+3|*OUT|' ('+2+','+3+')'|)SUB4,SUB3)
INPUT...(SUB4(&F2|1+$+$|)SUB1,SUB5)
INPUT...(SUB5(&A2|3+4|=&S|2+3+4|*OUT|' ('+2+','+3+','+4+')'|)SUB1,SUB5)
INPUT...
INPUT...(INITPRT(=*OUT|' '|*OUT|'FOLLOWING INITIALIZATION STATEMENTS GENERATED.'|)END)
INPUT...
INPUT...#  INITIALIZE CARRIES OUT THAT PROCESS, OPENING THE CALCOMP PLOTTER TAPE,
INPUT...#  AND ALLOCATING SPACE FOR COMS ARRAYS.
INPUT...(INITIALIZE(&F1|'GOES ON'+'GRAPHICAL OUTPUT'+'CALCOMP PLOTTER'|)INIT2,INIT1)
INPUT...(INIT1(=NEXT|'OPEN THE CALCOMP PLOTTER'|TMP|'COM1,EVL3,INIT2))'|)TMP)
```

```
INPUT...(INIT2(&F1|'ARRAY'+$|)END,INIT3)
INPUT...(INIT3(&A1|1|=ARANAM|1|TYPE|'NONE'|MODE|'REAL'|DIM|'(1)'|)END,INIT4)
INPUT...(INIT4(&F2|$+1|)INIT5)
INPUT...(INIT5(&A2|2|)INARA,INIT6)
INPUT...(INIT6(&F3|$+2|)INIT5,INIT7)
INPUT...(INIT7(&A3|3|=TMP|3|)INIT5,INIT8)
INPUT...(INIT8(TMP|'TYPE'|=TYPE|2|)INIT9,INIT5)
INPUT...(INIT9(TMP|'MODE'|=MODE|2|)INIT7,INIT5)
INPUT...(INARA(TYPE|'NONE'|ARANAM|$|=TYPE|2|)INAR1)
INPUT...(INAR1(TYPE|$|&F2|'FIRST DIMENSION OF'+$+1|&A2|2|=DIM|'('+2+')'|)INAR2,INAR1A)
INPUT...(INAR1A(=#TMP|'1DIM='+2|)INAR2)
INPUT...(INAR2(TYPE|$|DIM|'('+$+')'|&F2|'SECOND DIMENSION OF'+$+1|&A2|5|)INAR4,INAR3)
INPUT...(INAR3(=DIM|2+3+','+5+4|#TMP|'JDIM='+5|)INAR4)
INPUT...(INAR4(MODE|$|ARANAM|$|DIM|$|=NEXT|1+' '+2+3|TMP|'EVL3,INIT3))'|)TMP)
INPUT...
INPUT...(CMPRINT(=*OUT|' '|*CUT|'FOLLOWING INPUT IMPERATIVE STATEMENTS PROCESSED.'|)END)
INPUT...
INPUT...#  COMPUTE PROCESSES IMPERATIVE STATEMENTS INCLUDING COMMANDS.
INPUT...(COMPUTE(READ,GTNXT))
INPUT...(READ(*INPUT|'#'+$|=*OUTPUT|' '|*INPLT|2|)END,READ)
INPUT...(GTNXT(INPUT|$+';'+$|=NEXT|1|INPUT|3|)COMPUTE,EVSEQ)
INPUT...(EVSEQ(COM1,EVL1))
INPUT...(EVL1(NEXT|$' '+'END'+$' '|=*TMP|2|)EVLL,END)
INPUT...(EVLL(EVL2,GTNXT))
INPUT...(EVL2(NEXT|$' '+'DO '+$' '+$+'='+$+' TO '+$|=COV8|4|CCLM|8|DCXR|'1'|)EVL3,DODO)
INPUT...(EVL3(NEXT|$' '+'CALL '+$|=*NEXT|2+3|#NEXT|2+3|)EVL4,END)
INPUT...(EVL4(NEXT|$' '+$|=#NEXT|2|)EVL5)
INPUT...(EVL5(NEXT|$|=*NEXT|1|)END)
INPUT...(COM1(&F1|'COMMAND'+$|)END,COM2)
INPUT...(COM2(&A1|1|=TMP|''''+1+''''|)END,COM3)
INPUT...(COM3(TMP|$|NEXT|$' '+.1+$' '+$|=COMND|3|ARGS|5|)COM2,COMPRT)
INPUT...(COMPRT(NEXT|$|=*OUT|' '|*CUT|'FOLLOWING IS EXPANSION OF COMMAND...'+1|)COM4)
INPUT...(COM4(COMND|$|&F1|'COMMAND FOR'+1+$|&A1|1|=COMND|1|)COM5,COM4)
```

```
INPUT...(COM5(&F1|'PRCGRAM FOR'+$+1|&A1|1|=PRCGNAM|1|)CCM6,EXPAND)
INPUT...(COM6(&F1|'RESULT OF'+$+1|&A1|2|&F1|'GOES ON'+2+$|&A1|3|)CCMDFLT,CCM7)
INPLT...(CCM7(&F1|'PRODUCES&ON'+$+2+3|&A1|1|)COMFAIL,COM5)
INPUT...(COMDFLT(=RL|1|TMP|2|CUTPUT|'LINE PRINTER'|)CCMD1)
INPUT...(COMD1(RL|$|TMP|$|OUTPUT|$|)COM7)
INPUT...(COMFAIL(=*CUTPUT|' PRCGRAM FOR COMMAND '+1+' COULD NOT BE FOUND.'|)END)
INPUT...(EXPAND(ARGS|$+' '+$' '+$|=TMP|1|)EXPAN,EXP1)
INPUT...(EXPAN(ARGS|$|=TMP|1|)EXP1)
INPUT...(EXP1(TMP|$|PRCGNAM|$|&F1|$+1|)EXP1A,EXP2)
INPUT...(EXP2(&A1|3|)EXP1A,EXP3)
INPUT...(EXP3(&F2|'GRAMMAR RULE FOR&ON'+$+2+3|&A2|1|=TMP|1|)EXP2,EXP4)
INPUT...(EXP4(TMP|$+';'+$|=PD|1+','|SV|3+'.'|)EXP5,RSTRT)
INPUT...(EXP5(TMP|$|=PD|1+','|SV|'C.'|)RSTRT)
INPUT...(EXP1A(PROGNAM|$|&F1|'GRAMMAR RULE FOR'+$+1|&A1|1|=TMP|1|)EXPERR,EXP4)
INPUT...(EXPERR(=*OUTPUT|'NO GRAMMAR RULE NAME FOUND FOR '+1+'.'|)END)
INPUT...
INPUT...#  DODO PROCESSES 'DO' STATEMENTS FOUND BY COMPUTE.
INPUT...(DODO(DOLM|$+' BY '+$|=DOLM|1|DOXR|3|)DO1)
INPUT...(DO1(DOLM|$|DOXR|$|=#NEXT|4+5+6|#DOLM|1|DOXR|4+5+4+'+'+2|STMT|'0'|)DO2)
INPUT...(DO2(DOLM|$|=DOLM|'''+1+'''|)DO3)
INPUT...(DO3(DOTST,EXSAVD,ENDTST))
INPUT...(SAVE(=TMP|'=%SP'+1+'|'+'2'+'|'+')END)'|)TMP)
INPUT...(INCR(DOXR|$|=#TMP|1|)END)
INPUT...(ENDTST(DOVB|$|=#TMP|1|)ENDT1)
INPUT...(ENDT1(DOLM|$|TMP|$' '+.1+$' '|)CONTIN,END)
INPUT...(CONTIN(INCR,EXSAVD,ENDTST))
INPUT...(DOTST(STMT|$|=#STMT|'1+'+1|)DOT1)
INPUT...(DOT1(STMT|$|INPUT|$+'END OF LOOP'+$' '+';'+$|=SAV|6|ENDST|'''+1+'''|)DOT2,SAVE)
INPUT...(DOT2(STMT|$|INPUT|$|)DOT3)
INPUT...(DOT3(SAVE,READ,DOTST))
INPUT...(EXSAVD(=STMT|'0'|)EXLOOP)
INPUT...(EXLOOP(EX1,EVDO,EXTST))
INPUT...(EX1(STMT|$|=#STMT|'1+'+1|)EX2)
```

```
INPUT...(EX2(STMT|$|=TMP|'%SP'+1+'|'+'$'+'|'+'=INPUT'+'|'+'1'+'|'+')END)'|)TMP)
INPUT...(EVDO(INPUT|$+';'+$|=NEXT|1|INPUT|3|)END,EVD1)
INPUT...(EVD1(COM1,EVL3,EVDC))
INPUT...(EXTST(ENDST|$|STMT|$' '+.1+$' '|SAV|$|=INPUT|5|)EXLCOP,END)
INPUT...#THE FOLLOWING RULES CONVERT DO LOOP EXECUTION TO ALLOW FLOATING VARIABLES.
INPUT...(DC2(DCTST,EXSAVC,ENDTST))
INPUT...(ENDTST(DOVB|$|DOLM|$|=#TMP|'FLOAT(('+1+'-'+2+')-ABS('+1+'-'+2+'))'|)ENDT1)
INPUT...(ENDT1(TMP|'0.000000E+00'|)CONTIN,ENC)
INPUT...
INPUT...#  CLOSEOUT CLOSES THE CALCOMP TAPE, IF ANY.
INPUT...(CLOSECUT(&F1|'GOES ON'+'GRAPHICAL OUTPUT'+'CALCOMP PLOTTER'|)END,CLOS1)
INPUT...(CLOS1(=NEXT|'CALL ENDPLT'|)EVL3)
INPUT...
INPUT...# RULES FOR STOGRAM. THIS ROUTINE STORES C.F. GRAMMAR RULES IN THE ASSOCIATIVE
INPUT...# MEMORY, TO BE INTERPRETED LATER.
INPUT...(GRAMMAR(=*OUT|'FOLLOWING INPUT STORED AS GRAMMAR RULES.'|)STOGRAM)
INPUT...(STOGRAM(READ,XTORE,XTST))
INPUT...(READ(*INPUT|'#'+$|=*OUTPUT|' '|*INPUT|2|)END,READ)
INPUT...(XTST(INPUT|$+')'+$|=INPUT|3|*OUT|'END OF GRAMMAR.'|*OUT|' '|)STOGRAM,END)
INPUT...(XTORE(INPUT|$+'('+$+')'+$|=OUT|3|INPUT|5|)END,XTORT)
INPUT...(XTORT(OUT|$+'<'|INPUT|'>)'+$|=OUT|1+'<)>'|INPUT|4|)XTOR)
INPUT...(XTOR(X5,XTORE))
INPUT...(X5(OUT|$+'='+$|=N1|1|N2|'R5))'+3|)X1,X4)
INPUT...(X4(N2|'R5))'+$+'+'+$|=N2|'R1))'+2+','+4|)X3,X0)
INPUT...(X3(N2|'R5))'+$+'...'+$|=N2|'R2))'+2+','+4|)X2,X0)
INPUT...(X2(N2|'R5))<'+$+'>'+$|=N2|'R4))'+2|)X0)
INPUT...(X1(CUT|$+'*'+$|=N1|1|N2|'R3))'+3|)XERR,X0)
INPUT...(X0(N1|$|N2|$|=&S|1+2|)END)
INPUT...(XERR(OUT|$|=*OUT|'('+1+') IS AN ILLEGAL GRAMMAR RULE.'|)END)
INPUT...
INPUT...# THE FOLLOWING RULES GENERATE CHARACTER STRINGS WHEN GIVEN A GRAMMAR.
INPUT...(RSTRT(=OUT|''|)RNXT)
INPUT...(RNXT(PD|$+','+$|=NM|1|PD|3|)RDONE,GET)
```

```
INPUT...(GET(NM|$|&F1|1+$|&A1|2|=RL|2|)XVARB,RL)
INPUT...(R1(RL|$+'))'+$+',+$|PD|$|&F1|3+$|&A1|7|=NM|3|RL|7|PD|5+4+6|)ERR,RL)
INPUT...(R2(RL|$+'))'+$+',+$|PD|$+',+$|&F1|3+$|&A1|9|=NM|3|RL|9|PD|6+7+5+7+8|)ERR,RL)
INPUT...(R3(NM|$|SV|$+'.'+$|=NM|1+':'+2|SV|4|)ERR,GET)
INPUT...(R4(RL|$+'));'|CUT|$|=NEXT|3|RL|'EVL3,RSTRT))'|)R4A,RLTST)
INPUT...(R4A(OUT|$|RL|$+'))'+$|=CUT|1+4|)ERR,RNXT)
INPUT...(R5(RL|$+'))'+$|&F1|3+$|&A1|4|=NM|3|RL|4|)R5A,RL)
INPUT...(R5A(RL|$+'))'+$|=NM|3|)ERR,GET)
INPUT...(RLTST(PD|$+',+$|)END,RL)
INPUT...(RDONE(CUT|$|=NEXT|1|)END)
INPUT...(XVARB(NM|'.'+$|OUT|$|&F1|2+$|&A1|4|=CUT|3+4|)XVARB1,RNXT)
INPUT...(XVARB1(NM|'&'+$|XVARB2|$+'|''+$|=XVARB2|2+4+5|)ERR,XVARB2)
INPUT...(XVARB2(DUMMY|$|OUT|$|=CUT|2+1|)RNXT)
INPUT...(ERR(NM|$|PD|$|=*PD|'*ERROR* NM='+1+' PD='+2|)RNXT)
INPUT...
INPUT...# RULES FOR ANSWER. USES RULES FROM ASSERT.
INPUT...(ANSWER(READ,FIND,NTST))
INPUT...(NTST(INPUT|$+')'+$|=INPUT|3|)ANSWER,END)
INPUT...(FIND(INPUT|$+'('+$+')'+$|=OUT|3|INPUT|5|)END,FND)
INPUT...(FND(F5,FIND))
INPUT...(F5(OUT|$+',+$+',+$+',+$+',+$|)F4,END)
INPUT...(F4(OUT|$+',+$+',+$+',+$|)F3,AN4)
INPUT...(F3(OUT|$+',+$+',+$|)F2,AN3)
INPUT...(F2(OUT|$+',+$|)S1,AN2)
INPUT...(AN4(&F1|1+3+5+7|)FAIL,FOUND)
INPUT...(AN3(&F1|1+3+5|)FAIL,FCUNC)
INPUT...(AN2(&F1|1+3|)FAIL,FOUND)
INPUT...(FAIL(CUT|$+'$'+$|=*OUT|'THERE ARE NO ANSWERS TO ('+1+2+3+').'|)FCLSD,END)
INPUT...(FCLSD(OUT|$|=*OUT|'THE TRUTH OR FALSEHOOD OF ('+1+') IS UNKNOWN.'|)END)
INPUT...(FOUND(OUT|$+'$'+$|=*OUTPUT|'('+1+2+3+') HAS THESE ANSWERS:'|)CLOSED,FND3)
INPUT...(FND3(CUT|$+'$'+$+'$'+$+'$'+$|)FND2,AC3)
INPUT...(FND2(OUT|$+'$'+$+'$'+$|)FND1,AC2)
INPUT...(FND1(OUT|$+'$'+$|)CLOSED,AC1)
```

```
INPUT...(AC3(&A1|2+4+6|=*OUT|'('+1+2+3+4+5+6+7+')'|)END,AC3)
INPUT...(AC2(&A1|2+4|=*OUT|'('+1+2+3+4+5+')'|)END,AC2)
INPUT...(AC1(&A1|2|=*OUT|'('+1+2+3+')'|)END,AC1)
INPUT...(CLOSED(OUT|$|=*OUT|'('+1+') IS TRUE.'|)END)
INPUT...
INPUT...#  RULES FOR ASSERT.
INPUT...(ASSERT(READ,STORE,ATST))
INPUT...(READ(*INPUT|'#'+$|=*OUTPUT|' '|*INPUT|2|)END,READ)
INPUT...(ATST(INPUT|$+')'+$|=INPUT|3|)ASSERT,END)
INPUT...(STORE(INPUT|$+'('+$+')'+$|=OUT|3|INPUT|5|)END,STOR)
INPUT...(STOR(S5,STORE))
INPUT...(S5(OUT|$+',''+$+',''+$+',''+$+',''+$|)S4,END)
INPUT...(S4(OUT|$+',''+$+',''+$+',''+$|=&S|1+3+5+7|*OUT|'('+1+2+3+4+5+6+7+')'|)S3,END)
INPUT...(S3(OUT|$+',''+$+',''+$|=&S|1+3+5|*OUT|'('+1+2+3+4+5+')'|)S2,END)
INPUT...(S2(OUT|$+',''+$|=&S|1+3|*OUT|'('+1+2+3+')'|)S1,END)
INPUT...(S1(OUT|$|=OUT|1+'))'|)OUT)  THIS RULE ALLOWS EXECUTION OF OTHER ROUTINES
INPUT...
INPUT...# THESE EXTRA RULES MODIFY ANSWER TO LIST ANSWERS IN ENGLISH.
INPUT...(AN4(&F1|1+3+5+7|=AD1|'AD4'|)FAIL,FOUND)
INPUT...(AN3(&F1|1+3+5|=AD1|'AD3'|)FAIL,FOUND)
INPUT...(AN2(&F1|1+3|=AD1|'AD2'|)FAIL,FOUND)
INPUT...(END3(OUT|$+'$'+$+'$'+$+'$'+$|AD1|$|=AD1|8+',AC3))'|)END2,AC3)
INPUT...(END2(OUT|$+'$'+$+'$'+$|AD1|$|=AD1|6+',AC2))'|)END1,AC2)
INPUT...(END1(OUT|$+'$'+$|AD1|$|=AD1|4+',AC1))'|)CLOSED,AC1)
INPUT...(AC3(CUT|$+'$'+$+'$'+$+'$'+$|&A1|2+4+6|=OUTPUT|1+2+3+4+5+6+7|)END,AD1)
INPUT...(AC2(OUT|$+'$'+$+'$'+$|&A1|2+4|=OUTPUT|1+2+3+4+5|)END,AD1)
INPUT...(AC1(CUT|$+'$'+$|&A1|2|=OUTPUT|1+2+3|)END,AD1)
INPUT...(AD4(OUTPUT|$+','+$+','+$+','+$|=RLTN|1|SBJ|3|O1|5|O2|7|)END,W4A)
INPUT...(AD3(OUTPUT|$+','+$+','+$|=RLTN|1|SBJ|3|O1|5|)END,W3A)
INPUT...(AD2(OUTPUT|$+','+$|=PHRASE|1|SBJ|3|)END,WA)
INPUT...
INPUT...# THE FOLLOWING RULES GENERATE ENGLISH SENTENCES FROM S.T.L. N-TUPLES.
INPUT...(W4(O1|'.'|)W4A,END)
```

```
INPLT...(W3(SBJ|'.'|)W3A,END)
INPUT...(W2(SBJ|'.'|)WA,END)
INPUT...(WA(PHRASE|$|&F9|'NOUN'+1|=TMP|'ADET,WD))'|)WD,TMP)
INPUT...(WB(SBJ|'*'+$|PHRASE|$|=*CUTPUT|1+2+3+'.'|)WC,END)
INPUT...(WC(SBJ|$|PHRASE|$|=*OUTPUT|'*'+1+2+'.'|)END)
INPLT...(WD(PHRASE|$|=PHRASE|' IS '+1|)WB)
INPLT...(W3A(RLTN|'SUBSET OF'|C1|$|=PHRASE|2|MARK|'*ANY'|)W3B,W3A1)
INPUT...(W3B(RLTN|'DISJOINT FROM'|01|$|=PHRASE|2|MARK|'*NO'|)W3P,W3A1)
INPLT...(W3A1(MARK|$|SBJ|$|&F9|'NOUN'+2|=SBJ|1+' '+2|)W3A2,WA)
INPUT...(W3A2(=SBJ|1+'THING '+2|)WA)
INPUT...(W3P(RLTN|$|01|$|=PHRASE|1+' '+2|)W3C)
INPUT...(W3C(&F9|'FUNCTION'+1|=TMP|'TDET,WD))'|)W3D,TMP)
INPUT...(W3D(&F9|'NOUN ROOT'+1|=TMP|'ADET,WD))'|)W3E,TMP)
INPLT...(W3E(&F9|'VERB ROOT'+1|PHRASE|$|=PHRASE|' '+1|)WD,WB)
INPUT...(ADET(PHRASE|$1+$|&F9|'VOWEL'+1|=PHRASE|'AN '+1+2|)BDET,END)
INPUT...(BDET(=PHRASE|'A '+1+2|)END)
INPLT...(TDET(PHRASE|$|=PHRASE|'THE '+1|)END)
INPUT...(W4A(RLTN|$+'&'+$|01|$|02|$|=PHRASE|1+' '+4+' '+3+' '+5|)W4A1,W4B)
INPUT...(W4A1(RLTN|$|=*OUTPUT|'('+1+','+3+4+5+6+7+')'|)END)
INPUT...(W4B(RLTN|$|)W3C)
INPUT...(NOECHO)
```

References

Bobrow, D.G., 1964: "Natural Language Input for a Computer Problem Solving System," MAC-TR-1.

Feldman, J.A., 1965: "Aspects of Associative Processing," Lincoln Lab. Technical Note, MIT.

Forte, A., 1967: "SNOBOL3 Primer: An Introduction to the Computer Programming Language," MIT Press, Cambridge, Mass.

Gammill, R., 1966: "The Set Theoretic Language System," (a sequence of seven short papers written between May 1965 and February 1966).

Gammill, R., Marill, T., and Yates, J., 1967: "Relational Structures Research" Final Report of a Contract held by Computer Corporation of America.

Gammill, R., 1968a: "Final Report of DISHPAN Project," Project MAC Progress Report No. 5, MIT, Cambridge, Mass.

Gammill, R., and Gaut, N., 1968b: "A Graphical Input-Output Terminal Proposal," Internal Memorandum, Department of Meteorology, MIT.

Gammill, R., 1968c: "Contour Program" Application Program Manual No. 8, Applications Analysis Group MIT-IPSC.

Greville, T.N.E., 1967: "Spline Functions, Interpolation, and Numerical Quadrature," Mathematical Methods for Digital Computers, Vol. 2, John Wiley and Sons, New York, pp 156-168.

Halmos, P.R., 1960: "Naive Set Theory," D. Van Nostrand Co., Princeton, N.J.

Harrah, D., 1963: "Communication: A Logical Model," MIT Press, Cambridge, Mass.

IBM E20-0117-0: "Numerical Surface Techniques and Contour Map Plotting," IBM Data Processing Application, White Plains, N.Y.

Kaplow, R., Strong, S., and Brackett, J., 1966: "Map: A System for On-Line Mathematical Analysis," MAC-TR-24.

Poduska, J.W., 1966: "Notes on Searching and Sorting", Project MAC Memo-302, MIT.

Quine, W.V., 1959: "Eliminating Variables without Applying Functions to Functions, Journal of Symbolic Logic.

Raphael, B., 1964: "SIR: A Computer Program for Semantic Information Retrieval," MAC-TR-2.

Roos, D., 1967: "ICES System Design," MIT Press, Cambridge, Mass.

Stotz, R., and Cheek, T., 1967: "A Low Cost Graphical Display for a Computer Time-Sharing Console," ESL-TM-316, Electronic Systems Laboratory, Dept. of E.E., MIT, Cambridge, Mass.

Sutherland, I.E., 1963: "SKETCHPAD: A Man-Machine Graphical Communication System," Lincoln Lab. TR-296.

Teitelman, W., 1965: "FLIP - A Format List Processor," MIT Project MAC Memo M-263, Cambridge, Mass.

Van Horn, E.C., 1965: "Derivation of a Mean Pseudo-Associative Search Time," Machine Structures Group Memo. No. 13, Project MAC Memo-254, MIT.

Van Horn, E.C., 1966: "Hash-Code Searching," Machine Structures Group Memo No. 15, Project MAC, MIT.

Washington, W.M. et al., 1968: "The Application of CRT Contour Analysis to General Circulation Experiments, Bull. of the Amer. Meteor. Soc. Vol. 49, No. 9, pp. 882-888.

Weizenbaum, J., 1963: "SLIP: Symmetric List Processor," Communications of the ACM,Vol. 6, No. 9, pp. 524-544.

Welsh, J., 1967: "ANAL 68 Reference Manual," (a partially completed paper sent to this writer by its author). Travelers Research Center, Hartford, Conn.

Yngve, V.H., 1962: "An Introduction to COMIT Programming," MIT Press, Cambridge, Mass.

Yngve, V.H., 1960: "A Model and an Hypothesis for Language Structure," Technical Report 369, MIT Research Laboratory of Electronics, Cambrige, Mass.

Acknowledgments

Thanks are owed to so many, that some must be overlooked. The greatest helping of gratitude is due to Professors Edward N. Lorenz, Jack B. Dennis and James M. Austin, each of whom had occasion to rescue this work from an untimely end. Their encouragement and guidance were crucial to this interdepartmental endeavor.

Thanks also go to the many members of the Meteorology Department who have encouraged and supported the computer scientist in their midst. The MIT Computation Center and its programming staff were of immense help. All computations were performed there, and the staff spent many hours helping the author to learn the idiosyncracies of OS360.

Thanks are also due to my wife, Susan, whose patience and understanding knew no bounds, and to Mrs. Marie L. Gabbe who typed, and typed, and typed, without complaint.

Biography

Robert Gammill was born at Lincoln, Nebraska on February 12, 1938. He attended the University of Illinois Experimental High School and upon his graduation received a Bausch and Lomb National Science Scholarship to attend the University of Rochester. He received a B.S. degree with distinction in physics from that institution in 1959. Three years of service as a lieutenant in the Air Weather Service followed, the first year spent studying meteorology at Massachusetts Institute of Technology, the final two years spent as a computer programmer for the Global Weather Central at Strategic Air Command Headquarters in Omaha, Nebraska. Returning to M.I.T., Mr. Gammill completed an M.S. degree in meteorology in 1963. He then requested the formation of an interdepartmental committee to supervise his study of computer science. The present thesis is the culmination of that study.

At M.I.T. Mr. Gammill held the Space Technology Laboratories Fellowship and later served as a research assistant in the meteorology department. He is a member of the Association for Computing Machinery and of Sigma Xi.

His publications to date are:

Braille Translation by Computer, Report No. 9211-1, Department of Mechanical Engineering, M.I.T., October 1963, Contract No. SAV-1011-62, Department of Health, Eduction and Welfare.

The Man-Machine Relation in Meteorological Data Processing, S.M. Thesis, Massachusetts Institute of Technology, Department of Meteorology, June 1963.

Relational Structures Research, with Marill, T.M. and Yates, J.H., Final Report of Contract DA18-119-AMC-03049(X), Computer Corporation of America, Cambridge, Massachusetts, August 5, 1967.